University of Alberta,
Department of Computing Science

"MoBed": A Mobile Test Bed for investigating Web Access Solutions for J2ME™-enabled devices

by

Mildred N. Ambe ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements of the degree of **Master of Science.**

Department of Computing Science,
University of Alberta

Edmonton, Alberta
Spring 2004

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-612-96443-4
Our file Notre référence
ISBN: 0-612-96443-4

The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

Wireless devices like cell phones are popular in this day and age because they provide instant gratification and convenient services to users without restricting them to a particular place and time. Mobile devices support features like instant messaging, calendar services, and Web browsing. There is a growing need for Web access from mobile devices since numerous wireless applications require data from the Internet. Challenges arise when developing software for wireless devices, due to device constraints such as small screen sizes, limited memory, unreliable wireless connections and low processing power.

The goal of this research is to design a test-bed for investigating different caching and prefetching schemes for mobile devices that utilize the Java™ 2 Micro Edition platform (J2ME™); thereby opening up a fresh perspective for providing Web access solutions for small, wireless devices.

To my loving Mother and Father

Agnes and Cletus Ambe

Acknowledgements

I wish to extend my heartfelt thanks to my supervisors: Dr. Eleni Stroulia and Dr. Ioanis Nikolaidis. Their invaluable help, ideas and support through out my research pushed me on in the right direction. All their help is very much appreciated.

I would also like to extend many thanks to all my friends who supported me tirelessly through out my graduate studies. My special thanks go to Josephine Felix, Yiqiao Wang, Kavita Gandhi, Maurine Hatch, Edward Zadrozny and all others who offered me help, advice and fun conversations over a coffee.

I would like to thank Abdullah Tuncay for all his love and support and for possessing the knack of constantly reminding me to be happy and to keep a smile on my face.

Finally, I would like to thank my mother and father for all their love, prayers, and encouragement all through the years, especially for always reassuring me that I could succeed in all my endeavours.

Mildred N. Ambe
January 2004
Edmonton Alberta, Canada.

Table of Contents

CHAPTER 1 INTRODUCTION AND MOTIVATION 1
1.1 MOTIVATION 1
1.2 WHY J2ME? 2
1.2.1 Why bother with J2ME? 2
1.2.2 Basics of the J2ME architecture 4
1.2.3 Programming with J2ME 5
1.2.4 J2ME versus Other Java™ 2 Editions 6
1.3 CONTRIBUTIONS OF THIS RESEARCH 8
1.4 THESIS OUTLINE 9
CHAPTER 2 RELATED RESEARCH 10
2.1 WEB CACHING 10
2.2 TRANSCODING WITH DISTILLATION AND REFINEMENT 11
2.3 J2ME AND THE WEB 12
2.4 WEB ACCESS ON MOBILE DEVICES 12
2.5 WEB PREFETCHING 16
2.6 PERFORMANCE OF CACHING AND/OR PREFETCHING WITH PROXIES 21
CHAPTER 3 A CLIENT BASELINE ARCHITECTURE 24
3.1 ARCHITECTURAL DESCRIPTION 24
3.1.1 The Browser MIDlet 26
3.1.2 The HTML Parser Package 28
3.1.2.1 HTML Node object 30
3.1.2.2 The Parser 30
3.1.2.3 The HTMLReader class 31
3.1.2.4 The HTML Tag classes 31
3.1.2.5 The HTML Scanner classes 32
3.2 ARCHITECTURE IMPLEMENTATION 34
3.2.1 Drawbacks of using an emulated environment 36

3.3 MOBILE-RESIDENT BROWSER EVALUATION 36

3.3.1 The Dataset *37*

3.3.2 The Experiment *37*

3.3.3 Experiment Analysis *39*

3.3.4 Limitations of the Experiment *40*

CHAPTER 4 MOBED CLIENT-PROXY ARCHITECTURE 42

4.1 THE MOBILE CLIENT COMPONENT 43

4.1.1 The Browser MIDlet *45*

4.1.2 The Request Dispatcher *45*

4.1.3. The GUI Builder *46*

4.1.4 Adding a Cache on the Client *47*

4.2 THE PROXY SERVER COMPONENT 47

4.2.1 The Proxy Controller class *50*

4.2.2 The HTML Parser *50*

4.2.3 The Proxy Transcoder *50*

4.2.3.1 The Proxy Cache 53

4.2.3.2 Cache management 54

4.2.4 Session Tracker Engine *54*

4.2.4.1 The Prediction Engine 55

4.2.4.2 Prediction using Path Profiles 56

4.2.4.3 Generating Path profiles 56

4.2.4.4 Path Tree Construction from user sessions 57

4.2.4.5 Prediction using the Path Tree 60

CHAPTER 5 EMPIRICAL EVALUATION 63

5.1 EXPERIMENT 1: CACHING RESTRICTED TO THE PROXY LEVEL 64

5.1.1 Objective *65*

5.1.2 Workload Description *65*

5.1.3 Experiment Design *65*

5.1.4 Results and Evaluation *68*

5.1.5 Comparison to the Client Baseline architecture performance *69*

5.2 EXPERIMENT 2: DATA COMPRESSION USING THE PROXY TRANSCODER 70
5.2.1 Objective *70*
5.2.2 Workload description *71*
5.2.3 Experiment Design *71*
5.2.4 Results and Evaluation *71*
5.2.5 Comparison to the Client Baseline architecture performance *72*
5.3 EXPERIMENT 3: CACHING AT THE CLIENT-LEVEL WHILE PREFETCHING AT THE PROXY 73
5.3.1 Objective *73*
5.3.2 Workload description *74*
5.3.3 Structure of the Simulator *78*
5.3.3.1 Time sequence illustration of a simulation run 79
5.3.4 Object and Data Structures of the Simulator *81*
5.3.4.1 Client simulation on the MoBed Proxy server 82
5.3.4.2 The Simulator Control Flow 84
5.3.4.3 Experiment setup 87
5.3.5 Results and Analysis *89*
5.3.5.1 Experiment 3-1 89
5.3.5.2 Experiment 3-2 94
5.3.6 Comparison to the Client Baseline architecture performance *101*
5.4 SUMMARY 102
CHAPTER 6 CONCLUSION AND FUTURE WORK 104
6.1 RESEARCH CONTRIBUTIONS 104
6.2 FUTURE WORK 105
6.3 CONCLUSION 107
BIBLIOGRAPHY 108
APPENDICES 113
(A) THE MOBILE CLIENT COMPONENT 114
(B) THE PROXY SERVER COMPONENT 116
(1) Processing a request at the proxy 116

(2) Proxy Transcoder classes *118*

(3) Prediction at the proxy *119*

(4) Client Simulation at the Proxy server (Experiment 3) *122*

List of Tables

Table 1 - The J2ME MID Profile Packages 8

Table 2 - Characteristics of devices emulated using the J2ME Wireless Toolkit 34

Table 3 - Listing of Experiments performed using MoBed 63

Table 4 - Factor-level combinations for Experiment 1 67

Table 5 - Observed Proxy latency for all simulation runs in Experiment 1 68

Table 6 - A summary of the train/test sets generated from the C301 workload partitions 78

Table 7 - A summary of the train/test sets generated from the CS workload partitions ... 78

Table 8 - Setup for Experiments 3-1 and 3-2 88

Table 9 - Experiment 3: Factors and response variables 89

Table 10 - Experiment 3-1: Results obtained from the C301 workload 90

Table 11 - Experiment 3-1: Results obtained from the CS workload 91

Table 12 - Experiment 3-2: Results obtained from the C301 workload 96

Table 13 - Experiment 3-2: Results obtained from the CS workload 97

Table 14 - Prediction-accuracy rates observed from 2 different T-values using three C301 partitions (With a cache size of 8kB, and No Retraining phase). 99

Table 15 - Experiment 3-2: Prediction-accuracy rates observed from all four C301 partitions (with and without retraining). 100

Table 16 - Experiment 3-2: Prediction-accuracy rates observed from both CS partitions (with and without retraining). 100

List of Figures

Figure 1 - Java™ 2 Micro Edition MID Profile Architecture 5
Figure 2 - The Java Editions 6
Figure 3 - Web access in the Client Baseline Architecture 25
Figure 4 - Initial Browser Screen 26
Figure 5 - Browser screen with user input 26
Figure 6 - Requested Web content displayed 27
Figure 7 - Possible operations provided by the Browser 27
Figure 8 - Class diagram of the HTML Parser package 29
Figure 9 - An example showing the use of the HTML Parser object. 30
Figure 10 - Class diagram of the HTML tag classes (HTML Parser package) 32
Figure 11 - Class diagram of the HTML Tag Scanner classes (HTML Parser package) 33
Figure 12 - Association between an HTML tag class and its corresponding scanner class. 33
Figure 13 - Driver MIDlet used to initiate client requests from the mobile client, while gathering information on the heap size change over time 37
Figure 14 - Sample output from the Driver MIDlet 38
Figure 15 - Required heap size as a function of the number of 'parsable' HTML nodes. 39
Figure 16 - Time taken to fetch and render a requested page as a function of the number of 'parsable' HTML nodes. 39
Figure 17 - Interaction between main components in the MoBed Client-Proxy architecture 43
Figure 18 – Interaction between main sub-components in the Client Component 44
Figure 19 - A view of the Mobile Client Cache 47
Figure 20 - The Proxy server functionality using MoBed 49
Figure 21 - Mobile Browser Page Nodes 51
Figure 22 – The MoBed Proxy Transcoder functionality 52
Figure 23 - An example illustrating the addition of an element to the proxy cache 53
Figure 24 - An example showing the relationship between user sessions and path profiles 56

Figure 25 - PathTree construction algorithm that accepts a list of URLSequences (Algorithm extracted from [SKS98]). *59*

Figure 26 - Example showing how paths are maintained in a PathTree as path profiles. *59*

Figure 27 - Condensing path profiles *61*

Figure 28 - Predicting using Condensing path profiles *62*

Figure 29 - Experiment 1- Proxy location factor (Level 1): Remote proxy server *66*

Figure 30 - Experiment 1- Proxy location factor (Level 2): Proxy located on the Web server *66*

Figure 31 - Original bytes downloaded from Web servers Vs Proxy-transcoded bytes ... *72*

Figure 32 - Generating training and testing sets from a workload partition *76*

Figure 33 - A sample training set *77*

Figure 34 - A sample testing set. *77*

Figure 35 - Time sequence illustration of a simulation run *80*

Figure 36 - A URLUnit object *82*

Figure 37 - An illustration of the function of the Mobile Cache Manager. *83*

Figure 38 - Function of the Proxy MobileCacheManager component *84*

List of Abbreviations

APIs	*Application Programming Interfaces*
CLDC	*Connected Limited Device Configuration*
CDC	*Connected Device Configuration*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transport Protocol*
IP	*Internet Protocol*
J2ME	*Java II Micro Edition*
J2SE	*Java II Standard Edition*
J2EE	*Java II Enterprise Edition*
JVM	*Java Virtual Machine*
KVM	*K Virtual Machine*
LRU	*Least Recently Used (caching policy)*
LFU	*Least Frequently Used (caching policy)*
MIDP	*Mobile Information Device Profile*
PPM	*Prediction by Partial Match*
URL	*Uniform Resource Locator*
WWW	*World Wide Web*

Chapter 1 Introduction and Motivation

Mobile devices are becoming increasingly widespread and their everyday use is becoming indispensable. For this reason, a variety of software applications are being migrated to mobile platforms. These applications have to accommodate a range of constraints in contrast to their desktop counterparts, including a different set of interaction techniques, small screen size, limited memory and processing power. Furthermore, wireless web-based applications also have to deal with the unreliability of wireless connections due to possible disconnections, high bit error rate and low bandwidth. This chapter explains the motivation for this research and provides a brief introduction to the Java™ 2 Platform Micro Edition (from now on referred to as J2ME).

1.1 Motivation

J2ME is emerging as the de facto standard for handheld mobile devices and is being widely adopted as the platform for delivering web services to mobile users. This chapter introduces an experimental test-bed for evaluating caching and pre-fetching mechanisms for the J2ME platform.

Caching and prefetching are two common solutions for coping with low bandwidth and intermitent connectivity. Caching enables the storage of accessed web-based content in a local structure, anticipating similar future requests. Prefetching takes this idea one step further by anticipating future web-access by clients. There has been a substantial body of research on the performance of caching and prefetching mechanisms for wired network access. However, the problem is substantially different on wireless devices, due to the constraints faced by these devices.

MoBed provides an experimental test-bed for designing, developing and analyzing different caching and prefetching schemes that can be used in devising Web access solutions for J2ME-enabled devices. In the first stage of this research, a simple Web browser application was developed to fetch and display different types of web content such as static pages with text and images and dynamic forms. In this stage, the entire browser application functionality was implemented on the mobile device including URL fetching, web data retrieval, HTML parsing, and user interface generation to display of the parsed content on the mobile device. This approach proved to be very inefficient (as was expected) because of the limited device memory and the slow connections to the network. However, it provided a "reference" point against which other caching and prefetching schemes can be compared.

In the final stage of this project, a flexible client-server architecture was designed to replace the architecture described above. The client was resident on the mobile device and the server on a wired host acting as a proxy between the mobile client and the web servers. This new, flexible architecture allowed for various web-access functionalities to be flexibly distributed between the client, the proxy and the server, resulting in numerous possible configurations for experimentation. The ultimate goal is to readily provide Web access to wireless clients, while minimising delays. Using MoBed, the following scenarios for web access are investigated in the pursuit of this goal:

(a) Location of a local cache (at the client or proxy server)
(b) Caching using different policies and eviction schemes
(c) Prefetching data to the client, using user access history analysis
(d) Prefetching based on a 'prefetch request' signal from the mobile client

1.2 Why J2ME?

J2ME is emerging as a standard for handheld mobile devices and is being widely adopted as the platform for delivering web services to mobile users. J2ME is a version of Java targeting software development for smaller devices, such as Personal Digital Assistants, mobile phones, two-way pagers, etc. This section provides several reasons why J2ME is coming up on the forefront in the Wireless Web industry, and briefly describes the J2ME architecture (with an emphasis on the Mobile Information Device Profile).

1.2.1 Why bother with J2ME?

As the wireless Internet revolution continues to grow, mobile users expect more and more performance from handheld Internet-enabled devices. As these demands increase, more wireless application developers look toward using a programming language that is ideally suited for wireless devices, namely Java. Some of the major benefits of using Java as the programming language for wireless devices described by [RTV01] are outlined below:

- Java is rapidly becoming one of the most popular programming languages used by software developers worldwide; as such, there are numerous developers skilled in Java.
- Java has the advantage of being a modern object-oriented programming language, with better programming constructs and abstraction mechanisms than other tools and languages used for wireless software development.
- Java is cross-platform compatible: applications can be moved flexibly between different devices.
- Wireless Java technology allows for user interaction support and graphic capabilities for mobile devices.
- The Java platform allows for the dynamic delivery of content: Applications, services and content can be downloaded dynamically over different kinds of networks.

The J2ME platform addresses a range of devices from phones, pagers, to high-end devices like Internet TVs. As more consumers demand the services of such web-enabled devices, the result is an increased interest in J2ME. Some facts and statistics outlined below provide even more evidence of the fast growing base of J2ME:

- J2ME on handsets is supported by all major carriers and pushed by all major phone vendors (such as Motorola, Nokia, etc) [LG-J2ME].
- The Zelos Group (zelosgroup.com) is a provider of predictive analysis for technology vendors and service providers. It predicts that Java will be the dominant platform in the wireless sector, with support found in over 450 million handsets sold in 2007 [McA02]. It has also been observed that the interest in Java as a platform for mobile handsets has grown significantly, especially interest in services based on J2ME [McA02].
- J2ME on cell phones sells: [LG-J2ME] shows that this combination is a commercial success, with more than 94 million devices shipped worldwide since 2002 [LG-J2ME].

These are only a few of the statistics available that show the emergence of J2ME as a standard for the fast growing wireless web industry [LG-J2ME]. For these and many more reasons, research on wireless Interent usage using J2ME devices is rapidly increasing, especially since this platform 'has positioned itself as the best solution for an extremely wide range of small devices' [LG-J2ME]. As the wireless internet matures with time, research can bring us closer and closer to the goal of sophisticated Internet access for mobile users.

1.2.2 Basics of the J2ME architecture

J2ME was announced at the JavaOne Developer Conferences in June 1999 as a highly optimized Java run-time environment aimed at a wide range of smaller devices, such as pagers, Personal Digital Assistants (PDA), mobile phones and set-top boxes [Nyl01]. Due to the high level of diversity between these range of devices, 'an essential requirement for the J2ME architecture is not only small size but also modularity and customizability' [RTV01]. In order to achieve modularity and scalability, the J2ME environment provides a range of Java Virtual machines with different processor and memory capabilities to service the diverse range of devices supported.

There are two main types of concepts used in the J2ME environment: Configuration and Profile. A J2ME configuration defines a platform for a 'horizontal grouping of devices' [RTV01] outlining the features that are expected to be available on devices from the same category. A J2ME Profile represents a vertical device family to ensure interoperability within a certain vertical device family [RTV01]. A profile is layered ontop of a configuration, thereby extending the latter. As such, in a J2ME environment, 'an application is written for a particular profile, and a profile extends a particular configuration' [RTV01].

In this research, the focus is on low-end devices such as cell phones, pagers etc. This part of the J2ME environment consists of the Connected Limited Device Configuration (CLDC) that focuses on low-end consumer devices, and the Mobile Information Device Profile (MIDP). The MIDP resides on or extends the CLDC, which runs on top of Sun's KVM (K Virtual machine). The latter is a compact Java Virtual machine designed for mobile devices that are small in size and limited in resources.

Figure 1 below shows the relationship between the virtual machine, CLDC and MIDP in the J2ME environment.

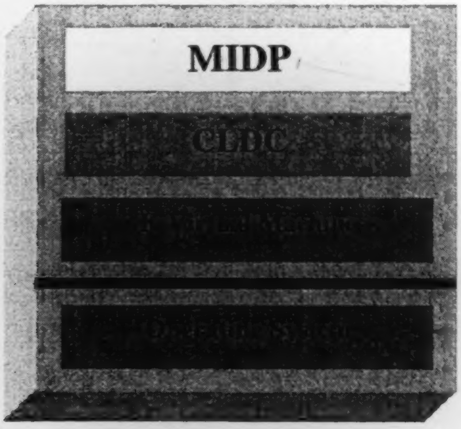


Figure 1 - Java™ 2 Micro Edition MID Profile Architecture

1.2.3 Programming with J2ME

The MID Profile is an extension to the CLDC, and hence inherits the CLDC Application Programming Interfaces (APIs). Java applications developed using MIDP are called MIDlets. They use only the APIs defined by the MIDP and CLDC specifications. A group of MIDlets of can be packaged and installed on a device in the form of a MIDlet Suite, and can be removed only as a group.

Other Java editions provide packages that target personal computers with adequate memory, disk storage, and processing power. J2ME targets low-end devices like cell phones and other devices with limited footprint that cannot possible handle big packages like the Java Standard and Enterprise editions. The MIDP package is considerably smaller in order to fit the restrictions of these devices. When developing applications in J2ME, there are some main issues that the programmer must be aware of:

- There is a growing number of devices from different manufacturers with different specifications that support J2ME, and the programmer has to be aware of what set of devices the application is targeting.
- J2ME devices have wireless networking, simple user interfaces and persistent storage for application-relevant data on the device. These properties differ from one device to the other, and the developer can choose to take advantage of them in different ways. In addition, mobile devices are always with the user, encouraging the development of mobile applications that can be customised to a user's needs.
- There are Java development tools available to the mobile Java developer such as Software development kits provided by some manufacturers, and Integrated Development Environments. Java SUN provides a J2ME Wireless Toolkit that provides examples, CLDC/MIDP documentation as well as a customizable environment for emulating the bahaviour of applications on a group of devices.

1.2.4 J2ME versus Other Java™ 2 Editions

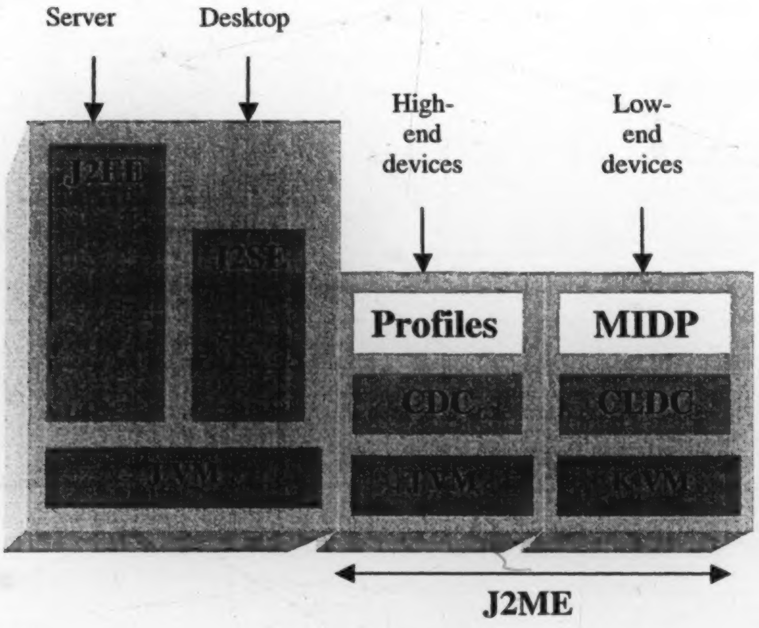


Figure 2 - The Java Editions

There are three main Java platforms available:

- Java 2 Enterprise Edition (J2EE) – Provides support for server-based applications
- Java 2 Standard Edition (J2SE) – Designed for desktops and personal workstations
- Java 2 Micro Edition (J2ME) – Designed for limited footprint devices.

Figure 2 above shows the different Java platforms in relation with each other. J2ME is a truncated version of J2SE. In order to keep the platform small and suitable for its target of small devices, a lot of the Java packages present in J2SE have been removed. Some of the J2SE packages that have been included in J2ME do not contain all the classes present in J2SE. Code written using J2SE can be run in J2ME only if the program uses Java API covered by both platforms. J2ME provides its own packages for persistent storage, user interface creation, and networking.

J2ME and J2EE can be brought together by means of mobile client-server applications. In such applications, the mobile client provides the user interface design, while the Server handles major computationally intensive tasks. Other J2EE platform functionalities can be used with MIDP clients, including Java Servlet API components, Enterprise JavaBeans components, XML, as well as JDBC (Java Database Connectivity) API.

The Proxy component classes in MoBed were implemented using the J2SE platform, while the Client component classes utilized the J2ME MIDP APIs. There are three main sub-components of the client component. First, the Browser MIDlet, which starts the web-browsing application. Second, the GUI Builder, which updates the user interface with the requested resource. Third, the Request Dispatcher, which manages the connection between the client and the proxy. These sub-components were implemented using some classes from all the J2ME MID Profile packages shown in Table 1 below. The Browser MIDlet and GUI Builder used some classes from the MIDP Core, Persistence, User interface and Application Lifecycle packages; and the Request Dispatcher used mainly classes from the MIDP Networking, Core and Application Lifecycle packages.

Table 1 - The J2ME MID Profile Packages

Package	API	Description
User Interface	javax.microedition.lcdui	Features for the implementation of user interfaces for MIDlets.
Persistence	javax.microedition.rms	Provides a mechanism for MIDlets to persistently store and retrieve data.
Networking	javax.microedition.io	Networking support based on the *GenericConnection* framework from the *CLDC*
Application Lifecycle	javax.microedition.midlet	Defines interactions between MIDlets and the environment in which they run.
Core	java.io, java.lang, java.util	System input and output classes; Language and Utility Classes included from J2SE

1.3 Contributions of this research

The main contributions of this thesis are outlined below:

- MoBed is used to investigate an intelligent method for flexibly combining caching and prefetching schemes towards providing Web access solutions for small, wireless devices. This project has achieved this objective while adaptively separating the mobile-resident from the proxy-resident functionality.
- J2ME is a fairly new specification that is rapidly growing in popularity. Such technology attracts research because it is still at an 'adolescent' stage, providing ample room for growth and improvement. This research introduces a fresh perspective on mobile web access targeted specifically towards the J2ME platform.
- Although numerous caching and prefetching schemes have been tested for wired-networked computers, there has been there has been no systematic study of the architecture and algorithmic design choices for mobile devices using J2ME. MoBed serves as testbed for studying caching and prefetching mechanisms specifically for wireless devices. Two experiments were conducted in this research. The first

experiment was designed to investigate the benefit of introducing caching at the MoBed proxy level, using two caching schemes: LRU and LFU. The second experiment consisted of trace-based simulations used to investigate a path profiling prefetching algorithm that predicts a user's next request based on past user access history, and the impact of having a cache on the mobile client. The main contribution of this research is not so much the results of these experiments, as the creation of a clean, modular, configurable architecture testbed design for investigating mobile Web access solutions.

1.4 Thesis Outline

This thesis is organised as follows. Chapter 2 provides a description of some related research in areas such as Mobile Internet access, Caching, Prefetching, proxy-based architectures, amongst others. Chapter 3 describes a Client Baseline Architecture, which demonstrates the implications of having a mobile-resident browser, where the mobile device performs computationally intensive tasks such as HTML Parsing, in addition to User interface generation. The improved version of the MoBed architecture is provided in Chapter 4. This architecture utilizes a client-proxy-server framework, which adaptively separates the mobile-resident from the proxy-resident functionality. Chapter 5 sheds more light on the MoBed architecture implementation and its validation by means of experiments and trace-driven simulations. A detailed description of the nature and analysis of these experiments is provided. The contributions of this research, as well as possible directions for future work are summarised in Chapter 6.

Chapter 2 Related Research

The related research discussions in this chapter are divided into the following topics: Web caching, Transcoding with distillation and refinement, J2ME and the Web, Web Access on Mobile devices, Web Prefetching, and the Performance of caching and/or Prefetching with proxies.

2.1 Web caching

Web caches exploit the locality of web page accesses, storing already-accessed web content in the hope that, when requested again, the transfer time from the server will be avoided, thus improving access time. By eliminating excessive transfers of stored web content, caches open the possibility of reducing bandwidth demands as well. However, for devices with restricted capabilities, one has to be aware that objects cached in their original form are not necessarily in the most "convenient" representation, as they will most likely need to be transformed in order to be rendered. In [KKO98], the authors address a set of modifications to classical proxy caching algorithms, which allow the implementation of a soft caching proxy system. They propose a strategy called 'Soft Caching', which allows a 'lower version' of a web object to become available on the proxy, in addition to the real object (anticipating different clients' needs). They provide a framework for the caching of images and media objects, but their focus is solely on the latter.

A more "classic" view is provided in [Dej99] where approaches to the issue of caching from the perspective of 'Temporal Locality' on the Web are studied, observing repeated accessed of one or more user to a single object. Given that caches are limited, in size, the caching policy used needs to have a reasonable eviction policy that determines which web objects are maintained in the cache or removed. The author's focus is on the estimation of probabilities for the prediction of the time of next access of the same web object. He proposed an algorithm that tracks the previous delays between accesses of a given web object, and uses these traces to determine the likelihood of that object being accessed again in the near future. Since this research is still in progress, there are no empirical evaluation results at the moment.

2.2 Transcoding with distillation and refinement

Incorporating a form of transcoding on web proxies has become an important subject of research as well. Transcoding allows data conversion from one format to one, which is better suited for the device that requested the information. A typical transcoder provides distillation, a process of compressing the data while still retaining enough semantic structure such that it is meaningful to the client [Cha95]. When a web document is distilled and sent to the client, the full version of the document is saved on the proxy to ensure that if the client requests any portion of the document (refinement), the proxy need not fetch the document from scratch. In this architecture, the proxy works with 'helper processes', which do the distillation using requirements provided by the proxy. Additionally, the author proposes a load balancing resource locator for proxies. It proposes an implementation of a complete prototype of the load-balancing resource locator on the UNIX platform. In a nutshell, a lightweight centralized server is maintained, which has the responsibility of managing proxies in a domain. Based on a client request, a proxy connects to the central server requesting a given transcoder, and the former allocates and caches transcoder addresses to the proxy. The central server performs load balancing by making intelligent decisions about which transcoder addresses to use [Cha95].

Additional research reported in [FB96] uses distillation and refinement of web data to bridge the gap between the low bandwidth client and high bandwidth of servers. An HTTP proxy was developed based on real-time distillation and refinement. In addition to the latter, statistical models were used, allowing the user to bound latency and exercise explicit control over bandwidth. The real-time data distillation eliminates the need for having several representations of a document, since desired intermediate representations can be created on demand using an appropriate distiller. The paper therefore claims that due to distillation and refinement, bandwidth is gained even if cycles are lost.

2.3 J2ME and the Web

Finally, J2ME is increasingly available on mobile device platforms being used to access web content. However, creating J2ME applications that interact with an enterprise server takes on interesting challenges that traditional client/server and browser-based applications do not face [Hem02]. Problems arise from the severely limited set of Java classes available to J2ME. Re-implementing certain classes from scratch to bring the level of support in J2ME to that of J2SE makes no particular sense due to the requirements for small memory footprint of the applications and the limited bandwidth constraint (in the event of loading code from elsewhere).

The Sun Microsystems white paper draft [SUN03] provides various new guidelines for designing wireless clients for enterprise applications using J2ME and J2EE technology. This will undoubtedly provide additional guidelines for Java developers interested in applications that use client-server architectures. We should point out that the platform restrictions result in a much more severe impact than deciding representations for the objects to be transferred. The platform effectively limits the expressiveness of the language due to the limited support classes. This latter aspect is, to the best of our knowledge, dealt explicitly for the first time as part of a proxy architecture.

2.4 Web Access on Mobile devices

There is a rising need and importance for wireless Web access from a wide range of mobile devices, from cellular phones, pagers, and in-car computers to palmtop computers and other small mobile devices. Nowadays, mobile devices contain features like email access, instant messaging, address book and calendar services, and Web browsing. Such devices are characterized by limited keyboard, small screens, low bandwidth connection, small memory amongst other constraints. Because of these constraints, small devices need special consideration when accessing information from Web servers. Mobile screen displays are generally much smaller than conventional personal computer screens, thus allowing for only a small amount of text to be displayed at a time. As such, there is a major issue with rendering Web content on mobile devices. Web pages may contain multiple images, search engines, forms, and other dynamic content, which cannot be displayed in the same manner as on regular workstations and desktops. In certain cases, the limitations of the device may result in certain content being non-renderable, or simply not accessible (e.g., navigation via image-maps on a low resolution device). These issues raise questions about different methods of improving mobile Web access using different languages, formats and architectural designs.

The authors of [CM03] propose an architecture called Scalable Browser for mobile devices. The Browser features include fetch-on-demand, progressive rendering, display on demand navigation style. The overall goal of the research is to enhance the user interface and browsing experience for handheld devices. The Scalable Browser architecture is based on a progressive delivery and rendering process whereby partial contents are rendered to the client. This is achieved by separating HTML pages into: structural data (which determines the style/geometric layout of HTML tree) and semantic data (descendants of structural data) [CM03]. In addition, the browsing is aided by converting the HTML pages to an intermediate SVG format (XML-based language), which retains all the features of HTML in order to ease the deployment process. The authors claim that their architecture retains the layout and rendering styles of the original document, reduces network overhead, improves legibility and provides a better interaction interface. Some limitations of this architecture are regarding the fetch-on-demand scheme, which may introduce additional latency between numerous fetches. In addition, when partial data is rendered, the user may request it by clicking on it. This action invokes a script resident on the client to completely render the content. This text formatting on the client may be computationally expensive for very small devices.

Research has also been done to analyze the browsing patterns of users on mobile devices, as is the case in [ABQ01]. The main goal of this research was to perform user-behavior analysis for mobile users to determine the following: the type of content wireless users are interested in; server loads over time; and the amount of time spent by wireless users on the channel while accessing content. The experiments for this research were carried out based on some analysis studies performed by monitoring the access activities for mobile clients on a site designed specifically for mobile usage. Firstly, the user behavior analysis was carried out by studying the distribution of wireless user sessions, as well as the number of bytes downloaded by each user. Secondly, the system load analysis study determined the Web server loads at different times in a day as clients accessed the system. Finally, a content analysis study was performed to determine the sort of content the mobile users are most interested in, such as yellow pages, entertainment etc. The authors show that they research has important implications by shedding some light on important issues for mobile access such as query caching, server scheduling, channel use, and TCP optimization.

A small device navigation model for web access is proposed in [STHK03], providing architecture allows existing WWW content and services to be used on wireless devices. The m-Links system is designed to achieved the following goals: web navigation on small devices, digging into embedded information on web pages for useful data, separation of service from links, and providing an open framework for others to develop services for wireless clients. One main advantage of this scheme is that the entire content from the requested site is not sent back to the client all at once. Pages are summarized in a neat, hierarchical format of links to enable clear navigation. A user is not flooded with the entire contents of a page at the initial step, but receives a list of links through which she can "dig" for more content ("dig and do" model) [STHK03]. The m-Links architecture consists of three main components: the link engine (processes web pages into link data structure); the service manager (returns services appropriate for each link e.g. read, print, send, etc), and UI generator (supports different user interfaces for different small devices). Although the m-Links architecture provides a new navigation model for small Internet device access, there is one main limitation the authors are currently working to improve – "link overload" [STHK03]. The latter describes a scenario where the model (being link-centric), reacts poorly when encountering pages with large number of links.

As discussed above, there may be a lot of benefits involved in summarizing or partially rendering web content to mobile users, instead of sending the entire web page contents to a small device. In [BGMP00], the authors introduce five methods for summarizing, browsing and progressively disclosing parts of web pages for small handheld devices. Using this scheme, the requested web page is divided into "Semantic

Textual Units" (STUs), which can be lists, paragraphs or image ALT tags (images are not displayed) that are arranged in a STU hierarchy. The main contributions of this research include: summarizing Web pages through partitioning into STUs and summarizing the parts. The authors developed and experimented with different summarization schemes involving selecting important, descriptive STU keywords. This summarization process is very important, as it is the core of the progressive disclosure mechanism used for mobile clients. Different keyword extraction techniques and summary-sentence extraction were performed to adequately summarize the STUs. The authors demonstrate with experiments that their summarization scheme in some case prove to be three or four times better than no-summarization schemes.

Papadopouli et al. [PS01] present a peer-to-peer data sharing system for mobile users called 7DS. The latter is a system that enables data exchange among peers (mobile or stationary) by operating in two main modes: prefetch and on-demand. When the mode is on prefetch, the system expects information needs of users and gathers resources by querying other peers. When the mode is on-demand, the user explicitly searches for desired information among its peers. 7DS reads a user's history file and predicts possible URLs the user may need later when there may be a loss of Internet connection. In this architecture, 7DS clients store data, URLs, web pages and exchange them with interested peers.

Sabnani [Sab97] shows that proxies can be used to process control information and manipulate data exchanged between the mobile client and server. The author discusses some benefits of proxies for mobile Web access: (1) proxies may hide the diversity of mobile devices from applications; (2) proxies may reduce the amount of communication involving the mobile device, hence reducing the consumed air bandwidth; and (3) proxies may take over the execution of complex functions, freeing resources of the limited devices. All these advantages are the main reasons why MoBed involves a client-proxy-server architecture.

2.5 Web Prefetching

As WWW resources and services increase, significant delays are introduced in the form of network latency, server overloads and slow response times. Prefetching has been introduced as one of the potential schemes for reducing this Web latency. Many different prefetching models have been introduced for usage on wired networks, but not on wireless networks. This section sheds some light on some of the different prefetching models that have been researched in the past.

Many prefetching models prefetch web pages based on user profiles. The research conducted in [HBA99] introduced a prefetching model which studies the Web page contents for all users, builds up a user profile and knowledge base from this content analysis, thereby recognizing users' individual preferences. This model combined caching and prefetching in a proxy server (proxy-initiated prefetching). The latter maintained user access logs and user interest profiles over periods of time. These two profiles were used for creating user interest registers, which provide an insight on the user's possible interests (obtained by keyword extraction from visited pages). A user's profile is constantly updated by means of a 'recency-before-frequency selection algorithm' [HBA99], and based on the user's interest profile; web pages are prefetched for the client. As such, this study focused on users' individual preferences as the basis for prefetching.

In another recent study by Cunha and Jaccoud [CJ97], the authors developed two user models that can be used in conjunction with prefetching schemes. The first model uses Random Walk Approximation (for capturing the long-term trend) and the second based on Digital Signal Processing (DSP) techniques (for short-term behavior). A user's navigation strategy is taken into account as it poses a challenge in prefetching procedures. Some users surf the Net over multiple, while others are more inclined to access pages on the same site. User profiles can be explored by means of the two user models. For more information on the details of the two empirical user models, see [CJ97]. These models can be used for predicting a user's next move, in combination with a prefetching procedure.

Nanopoulos et al. in [NKM01] adopted a data mining approach to prediction for WWW access. The authors propose a method based on association patterns, which consider all the main features of Web user navigation. One of the main contributions of this paper includes the identification of three main factors affecting Web prefetching: the arrangement of page accesses, the noise present in access sequences, and page access dependencies. A new prefetching algorithm, *WMo* is also introduced and proven to outperform existing algorithms. The WMo algorithm focuses on preserving ordering in the access sequences, which is important for prefetching. All the details of the candidate generation and pruning processes of WMo are covered in [NKM01]. The performance of WMo was evaluated using a synthetic data generator, and analysed to achieve high prediction rates.

Although caching and/or prefetching provide some degree of latency reduction for Web users, all the latency cannot be completely eradicated due to bandwidth restrictions, and download bottlenecks. Kroeger et al. [KLM97] explore this further by investigating an upper bound for proxy-based caching and prefetching as means to effective Web latency reduction. By distinguishing caching and prefetching algorithms into distinct categories, the authors introduced four different models by which to test for upper bounds. The experiments and simulations from this research show that caching and prefetching can indeed reduce latency even though these techniques have limits in their ability to reduce latency. A combined caching and prefetching proxy was shown to be able to reduce latency by 60% at best [KLM97].

Prefetching in the Web can be initiated in most cases by three entities: a server communicating with a client, an intercepting proxy, or a client machine. A combination of client and proxy or client and server can be involved in the prefetching process. In [PM96], Padmanabhan and Mogul introduce a scheme where a server sees several requests from multiple clients, make predictions for pages, informs the client of the files that may be prefetched and lets the client make a decision on whether to prefetch the files or not. This decision is based on the clients' current disposition such as memory, bandwidth, or cache storage issues. In this case, even though the server determines what files are possible options for prefetching, the client initiates prefetching. The prefetching algorithm on the server represents patterns of accesses by creating dependency graphs stored locally, and which are dynamically updated as new requests are satisfied. When a page is being accessed, another page is only considered for prefetching if the weight of the arc between the two pages (on the dependency graph) exceeds a prefetch threshold. In general, the results of this research demonstrate that even though predictive prefetching can significantly reduce perceived latency, there is a trade-off in increased network traffic.

Prediction by Partial Matching (PPM) is a widely used prediction algorithm employed in Web prefetching. PPM algorithms perform predictions from a prediction tree obtained from historical URLs. Chen and Zhang [CZ02] introduce a variation to PPM by incorporating popularity information into the PPM model (popularity-based PPM). The authors define the popularity of a URL as the number of times it is accessed in a time period. Incorporating popularity information into the PPM model involves altering the prediction tree as follows: branches that hold popular URLs can lead a set of long branches, while less popular documents can lead a set of short branches. In addition, this model implements optimization alternatives to reduce space allocation. Through trace simulations, the authors show that their popularity-based PPM model greatly reduces storage space for tree nodes; and outperforms other techniques by 5 to 10% of hit ratios.

Many prefetching techniques make predictions based on user profiles and user's history, but B. Davison [Dav02] examines the technique of prefetching a user's next move by analyzing the content of recently requested pages from the user. He shows that this approach can make predictions of actions that the user may never have done in the past (as opposed to prefetching based on historical references). The goal of the research is to improve user-perceived performance without querying the user for interest topics, or altering pages presented to the user. Their approach involves modeling a user's changing interests by analyzing the textual contents of pages recently requested by the user. A total of four methods were presented: baseline random ordering, original rank ordering, and two others which rank URLs based on the similarity of the link text and non-HTML text of preceding pages [Dav02]. The results from his research show that similarity-based rankings performed 29% better than random link selection methods, and 40% better than no prefetching at all (with infinite cache). The results also show that approximately 40% of the time, a user requests information that has never been seen before; hence showing the importance of prefetching based on page content as opposed to past accesses.

P. Cao et al. [CFK95] presents the integration of caching and prefetching as effective techniques of improving the performance of file systems. It provides a performance evaluation (by simulations) of two prefetching strategies: aggressive and conservative strategies. These strategies address the interaction between caching and prefetching, while tackling the main issues like when to start prefetching, what should be prefetched and what should be thrown out. The aggressive prefetching strategy always prefetches the next block into the cache at the earliest opportunity that is presented. The conservative prefetching strategy minimizes the elapsed time while performing a minimal number of fetches. Their simulations show that the two strategies are close to optimal while reducing the application elapsed time by 50%. This study focused on integrated caching and prefetching for file systems, as opposed to the WWW scenario, but provides insight on the integration of the two schemes.

The study in [CZ01] shows the importance of Web server input in the process of proxy-based prefetching. It proposes a coordinated proxy-server prefetching technique that coordinates prefetching activities at the web servers and proxy. In this paper, the authors investigated the shortcomings of proxy-based prefetching to discover situations where help is needed from web servers. This study employs the PPM (Partial Match) prediction technique for prefetching. Prefetching starts after a requested object is accessed at a level of the Internet caching system. The process continues by a search of the PPM tree rooted by the requested object. A relative probability is assigned to every object in the PPM tree as a ratio between the number of accesses to that object versus the number of accesses to the root object. This relative access probability is the variable used for adjusting the prediction accuracy in both proxy and Web servers (set to 30% in this paper). If the requested object is not found in the PPM tree, the Web server makes the prefetching decision. The coordinated proxy-server prefetching techniques was evaluated by measuring the reduction in communication over the server-based approach, and by comparing its hit ratio with those of the server and proxy based methods. Their results show that the byte hit ratios and hit ratios from the coordinated proxy-server prefetching are up to 88% higher than from proxy-based, and comparable to server-based prefetching results with 5% difference.

Markatos and Chronaki [MC98] present a Top-10 approach to prefetching, whereby there is a combination of client access profiles with the servers' knowledge of their Top-10 most popular documents. The servers periodically calculate their top-10 most popular documents and prefetch them only to clients that can potentially use them (i.e. to 'frequent' as opposed to 'occasional' users). The Top-10 approach is based on a client-proxy-server framework. On the server end, the Top-10 daemon processes the user access logs, computes the list of ten popular documents on that server and updates a web page showing this information, served by an HTTP server. On the client end, there resides a prefetching agent, which gathers a log of all the HTTP requests of the client and periodically creates a prefetching profile of the client (list of potential servers for prefetching). Based on this prefetching profile, the prefetching agent requests the most popular documents from the activated servers. From trace-driven simulations based on access logs from different servers, the performance results of the Top-10 approach in [MC98] show that the Top-10 scheme can anticipate more than 40% of a client's requests with a network traffic increase of 10 to 20%.

Duchamp [Duc99] examines a new method for prefetching Web documents into the client-side cache. The clients send references to Web servers, which collect the information and disperse it to other clients. The reference information indicates how often hyperlink URLs embedded in pages have been previously accessed relative to the embedding page. The clients initiate the prefetching using any algorithm, based on their general knowledge of the popular hyperlinks. Dynamically generated pages, as well as pages with cookies can also be prefetched using this scheme. In addition, the prefetching algorithm measures the available bandwidth to the client and limits the prefetching requests to only a fraction of the bandwidth available. The author shows that as a result of these measures, prefetching is improved, client latency is reduced by 52.3%, the prefetch accuracy is 62.5% (prefetched pages that are eventually used), and network traffic less than 24%.

2.6 Performance of Caching and/or Prefetching with Proxies

M. Abrams et al. [ASA+95] conducted a study to assess the limitations and potentials of proxy servers in the caching of Web documents retrieved with different protocols using WWW browsers. The research used a cache simulation with traffic corresponding to three educational workloads over a semester period. The experiments involve examining and comparing three different cache replacement policies: LRU (classic), LRU-MIN (variant of LRU that minimizes number of replaced documents), and LRU-THOLD (variant of LRU where no document larger than a threshold is cached). This study demonstrated that even though caching provides valuable benefits by lowering traffic and bandwidth, it does not provide a complete solution to the Web latency problem. The results from this study show that: (1) using their workloads, the maximum possible hit rate for a proxy is 30 to 50%; (2) the classic LRU policy is a poor option when the cache is full and a document replaced, even though simple variants can drastically improve the results; (3) some modifications to proxy server configuration parameters for a cache may provide little benefit; (4) with the workloads used in this research, the proxy cache hit rate tended to decline over time; and (5) hit rates increased up to 20% when all documents are cached, regardless of its domain.

Padmanabhan and Mogul [PM96], in their research paper on using predictive prefetching as a means to improve WWW latency, showed clients that access the WWW through proxy caches may reap some benefits. They showed that prefetching could take place in two main ways: between Web servers to proxy and between proxy cache to clients. It was observed that proxies are in a good position to make prefetching-related decisions relative to Web servers, since they can observe different client access patterns across servers. The authors also outlined two main situations in which a proxy cache can be invaluable from the point of view of prefetching: (1) In a scenario where a client is connected to a proxy via a non-shared line, the idle time observed could be filled up with prefetch traffic while ensuring that other traffic flow across the connection is not hindered; and (2) in a second scenario where each client gets data through a high-latency, high-bandwidth connection and the reverse connection may be through a slow line. In this case, prefetching would be an attractive option when the spare bandwidth and large startup latency of fetching data on demand are taken advantage of. The authors showed that the two scenarios described above could prove to be especially useful for prefetching.

Markatos and Chronaki in [MC98] present a Top-10 approach to Prefetching on the Web, and show that the use of proxies could have a positive effect on the performance of prefetching. The authors advocated the use of proxies by showing that, in the event of all clients accessing a server through a proxy, the latter could aggregate all clients' requests and qualify for prefetching as a repeated and heavy client [MC98]. They also mention that the proxy could prefetch documents on behalf of any of its clients. This activity would improve performance since a number of clients could be interested in the same document that was prefetched once. The authors used trace-driven simulations with artificial proxies, in order to test the effect of proxies on prefetching. Their experiments showed that the hit ratio using proxy servers doubled or tripled compared to hit ratios with no proxies used. They also observed that the increased hit ratios came with almost no increase in network traffic; in some cases, the traffic increase was less than 20% and in some occasions, there was a decrease in traffic. The reason for this decrease was analyzed and determined to be as a result of many clients using the same prefetched document. The research was taken one step further with the study of the effect of second-level proxies on prefetching. Second-level proxies aggregate requests from first-level proxies (which get requests from user-level clients) [MC98]. Their results for prefetching were even more promising in this case. One server used in the experiment reached a hit ratio of over 60%, showing a marked performance improvement, usually accompanied by a decrease in traffic. Their research shows that proxies have an impact on the process of prefetching documents from the WWW to clients.

Chen and Zhang in [CZ01] propose a coordinated proxy server prefetching technique that uses reference access information and coordinates prefetching at the proxy and Web servers. This research also involved the investigation of the conditions that make proxy-based prefetching ineffective and needs help from Web servers. The authors defined the relative hit ratio for a proxy as 'the ratio between a hit ratio from the proxy-based prefetching and the hit ratio from a server-based prefetching' [CZ01]. It was observed that as the number of clients accessing the proxy server increased, the proxy-based prefetching ability based on relative hit ratios with a given server increased as well. More precisely, in the conducted experiments, it was observed that as the client count increased from 16 to 64 users, the average relative hit ratios increased from 59% to 79% (for all servers used). Another important dimension in this research was to determine the shared request distribution to different servers through the proxy in a WWW environment. This study showed that in a proxy server with 1000 clients, proxy-based prefetching could satisfy less than 40% of requests (with about 60% relative hit ratio). In addition, for over 60% of the requests, prefetching at the proxy may not be sufficiently adequate. The authors hence show that proxy prefetching is limited, as the other studies described above show.

Although a lot of work has been done in caching using web proxies that act as intermediaries between the client and the servers, few studies address the specific topic of caching content (possibly in a transformed representation) that can be useful to mobile devices. The related work discussed in this chapter was divided into the following categories: WWW caching with/without proxies, transcoding with distillation and refinement, J2ME and the Web, Web access from Mobile devices, prefetching schemes for wired/wireless WWW access, and the performance of caching and/or prefetching with proxies. Although numerous caching and prefetching schemes have been tested for wired-networked computers, there has been there has been no known systematic study of the architecture and algorithmic design choices for Web access in mobile devices using J2ME.

Chapter 3 A Client Baseline Architecture

This chapter provides a detailed description of the first stage of this research. In this stage, a mobile client baseline architecture was developed to provide a "reference" point against with other architecture designs could be compared in the later stages of this research. The idea for this architecture originated with an interest in porting a Web-based browser application (that would be very practical for a workstation using on a wired network), to a small mobile device that relies on a less reliable, wireless connection to the Web. This architecture shows that by simply porting such an application to a wireless client does not provide an adequate or practical solution to Web access for wireless devices, due to the intense level of processing that is required to sufficiently run the application. The costs of operating a simple Web browsing application (in terms of memory and time costs) are extremely noticeable on a constrained mobile client, as opposed to a wired client (like a personal computer) that has much larger memory capacities and better connectivity to the Web.

Sub-sections 3.1 and 3.2 contain an in-depth description of the baseline architecture and its implementation; and the remaining sub-sections describe the evaluation process of this architecture

3.1 Architectural description

The overall web access process in the Client Baseline Architecture is illustrated in Figure 3 below. The architecture consists of the following components:

- The *Browser MIDlet* or User Interface Manager, which initiates, creates and manages the mobile user interface.
- The *HTML Parser*, which establishes a connection with the Web server from which the request can be satisfied, and provides the HTML parsing capability for converting the requested resource data to a client-friendly version.

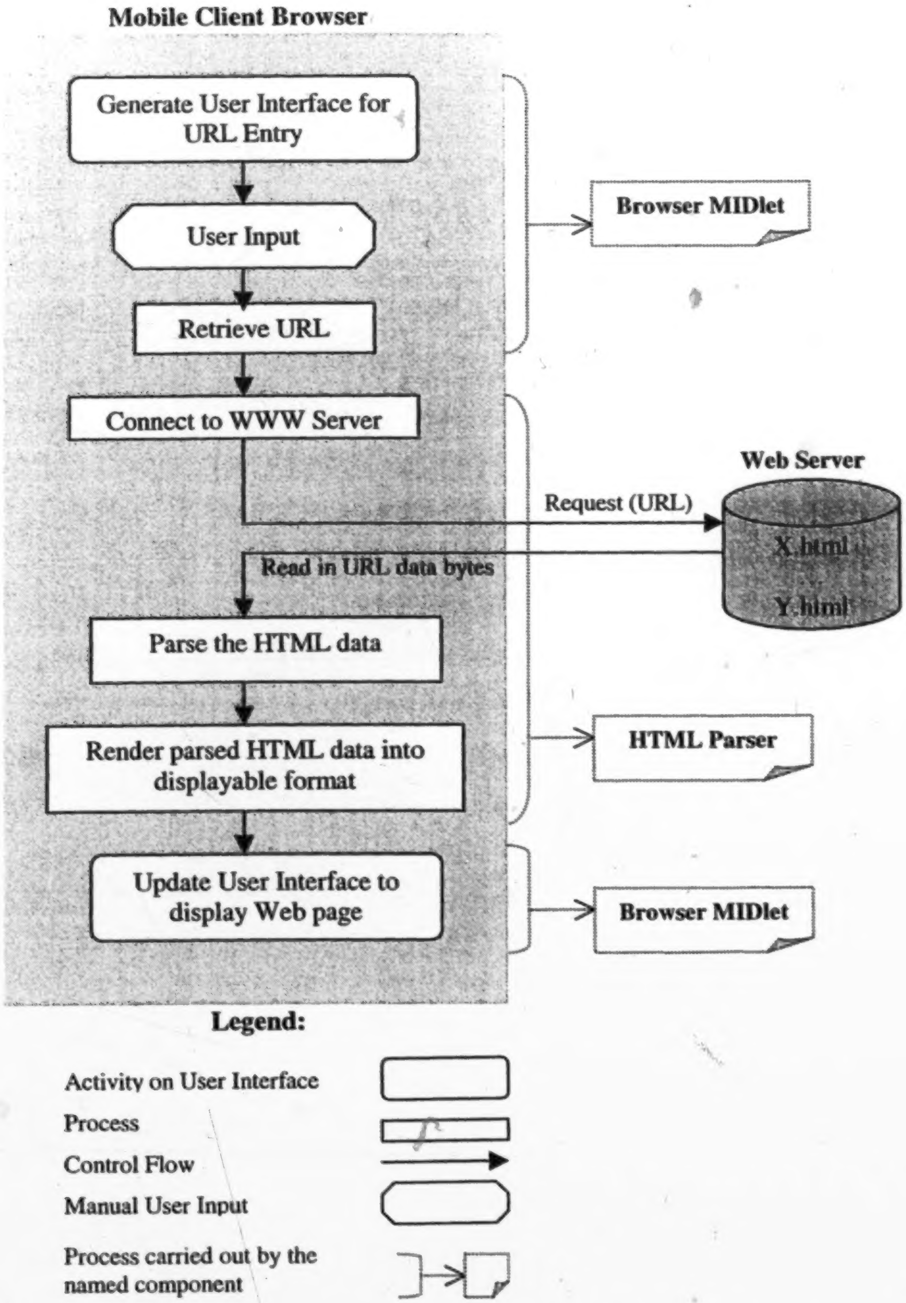

Mobile Client Browser
Generate User Interface for URL Entry
User Input
Retrieve URL
Connect to WWW Server
Browser MIDlet
Web Server
Request (URL)
X.html
...
Y.html
Read in URL data bytes
Parse the HTML data
Render parsed HTML data into displayable format
HTML Parser
Update User Interface to display Web page
Browser MIDlet
Legend:
Activity on User Interface
Process
Control Flow
Manual User Input
Process carried out by the named component

Figure 3 - Web access in the Client Baseline Architecture

3.1.1 The Browser MIDlet

When the mobile client application is started, the Browser MIDlet initiates the user interface by displaying a simple form allowing the client to enter the URL requested, as shown in the snapshots in Figure 4 and Figure 5 below.

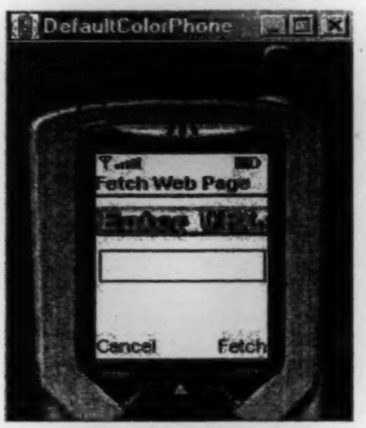
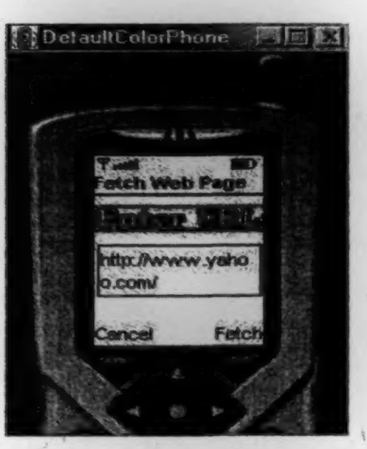

DefaultColorPhone
Fetch Web Page
Enter URL:
Cancel
Fetch

Figure 4 - Initial Browser Screen


DefaultColorPhone
Fetch Web Page
Enter URL:
Cancel
Fetch

http://www.yahoo.com/

Figure 5 - Browser screen with user input

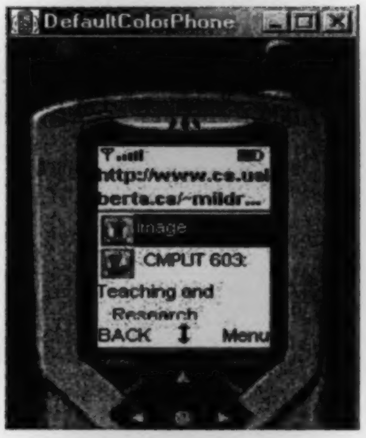

DefaultColorPhone
http://www.cs.ualberta.ca/~mildr...
Image
CMPUT 603:
Teaching and
Research
BACK
Menu

Figure 6 - Requested Web content displayed

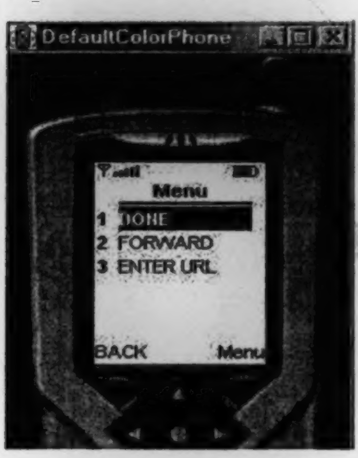

DefaultColorPhone
Menu
1 DONE
2 FORWARD
3 ENTER URL
BACK
Menu

Figure 7 - Possible operations provided by the Browser

After the resource requested is retrieved from the Web Server and parsed, the user interface is updated with the Web content. The Browser MIDlet 'renders' the content on the device by outputting the displayable information such as links, texts, and images on the screen in the form of a list (as shown in Figure 6 above). Using the navigation buttons on the mobile phone, the user can navigate through the contents of the list, and click on any links for viewing. This browser does not display embedded images on web pages once they are initially rendered. If the image icon on the page is clicked on, the image can then be viewed. The visited URLs can be navigated by means of back and forward soft buttons on the mobile phone. Figure 7 illustrates the different operations that can be carried out on the Browser when a user is perusing the content. The browser MIDlet also handles dynamic content and user interaction though forms, as well as maintains sessions and cookie management. When a user submits a form, a POST HTTP request is made to the server, and the response (if returned in HTML) is retrieved, parsed and rendered on the screen.

3.1.2 The HTML Parser Package

An open source HTML parser developed by the Kizna Corporation [HP1.1] provided the HTML parsing capability needed for the mobile Web access. This parser was chosen for use in this project for the following reasons. Firstly, it is implemented in Java using the Java 2 Standard Edition. Secondly, the source code had a minimal use of classes and packages that are absent in J2ME specification. This meant that there would be few changes needed to adapt the parser in order to make it functional in J2ME. Thirdly, the source code is open for public use and can be re-used or modified by the programmer. The code was adapted in order to make it usable in a J2ME/MIDP environment, by the addition of some functionality to the original classes, and providing new classes to augment the HTML-parsing ability. Figure 8 below provides a simplified class diagram showing the classes and directories found in the HTML parser package. Note that the class diagrams provided in this section contain only the interesting attributes pertaining to a given class, such as the major class members, parameters and methods.

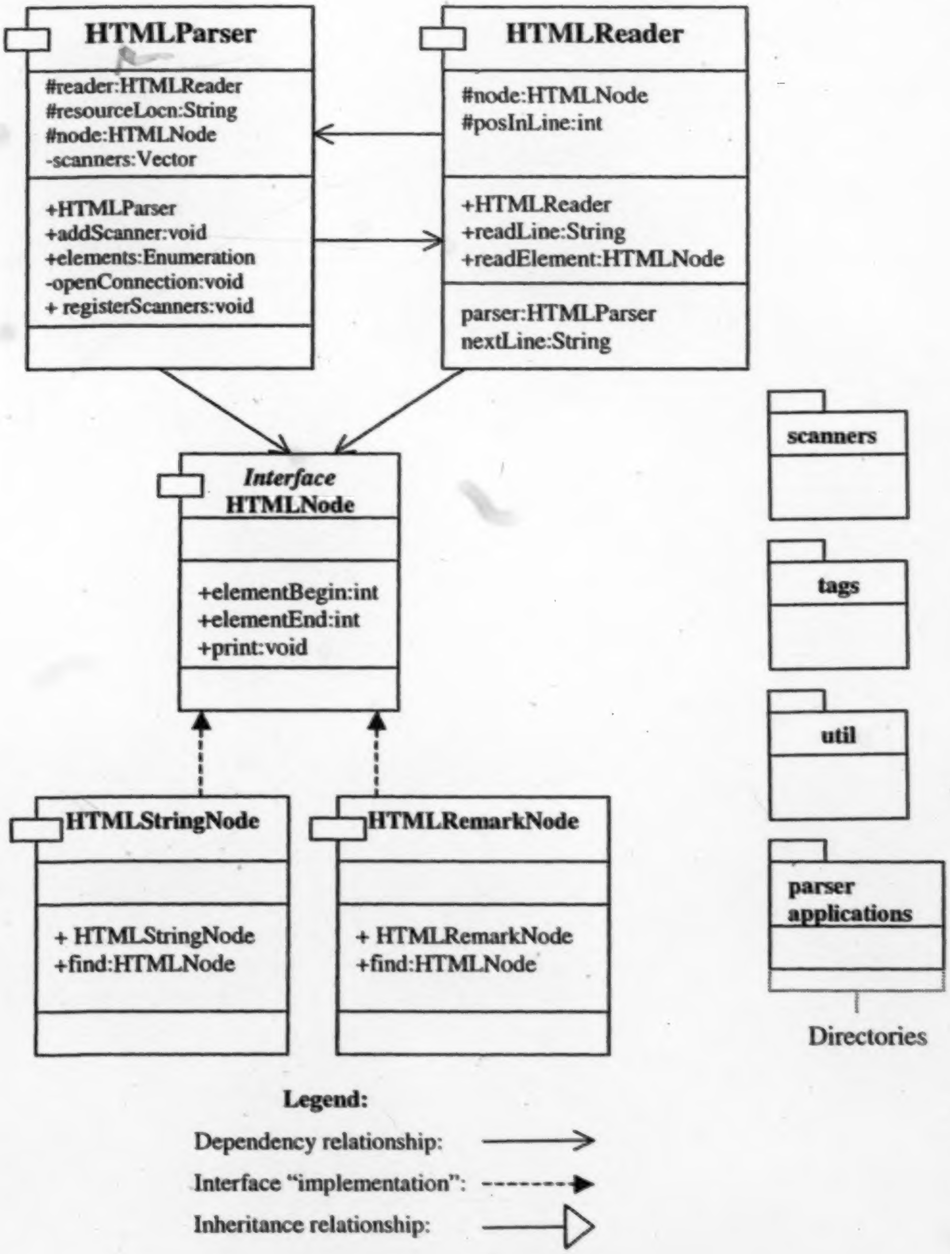

HTMLParser
#reader:HTMLReader
#resourceLocn:String
#node:HTMLNode
-scanners:Vector
+HTMLParser
+addScanner:void
+elements:Enumeration
-openConnection:void
+ registerScanners:void
HTMLReader
#node:HTMLNode
#posInLine:int
+HTMLReader
+readLine:String
+readElement:HTMLNode
parser:HTMLParser
nextLine:String
Interface
HTMLNode
+elementBegin:int
+elementEnd:int
+print:void
HTMLStringNode
+ HTMLStringNode
+find:HTMLNode
HTMLRemarkNode
+ HTMLRemarkNode
+find:HTMLNode
scanners
tags
util
parser applications
Directories
Legend:
Dependency relationship:
Interface "implementation":
Inheritance relationship:

Figure 8 - Class diagram of the HTML Parser package

3.1.2.1 HTML Node object

The HTML Node class is an interface implemented by all types of nodes, such as strings, and all kinds of tags. The HTML Tag class and all its subclasses implement this interface. For example, an HTML Image tag is an HTML node, even though it may contain different information from say, a Link or Form Tag. Different nodes hold information relevant to the tag they represent. For example, an Image tag contains the resource link for the image, and its textual description.

3.1.2.2 The Parser

This class is used either to iterate through HTML page or to directly parse the contents of the page and print the results. Typical usage of the scanner involves the following steps: The Parser object is initialised with the requested URL. All the tag scanners that pertain to the HTML tags to be parsed are registered (scanners are described in Section 3.1.2.4 below).

By calling the parser object's *elements()* method, parsing occurs on demand. The Parser object initializes an *HTMLReader* object, which provides methods to read in data from the source. It is important to note that the parsing occurs only when the parser object is enumerated, by calling its *elements()* method. Figure 9 shows a sample code depicting the use of the HTMLParser object. The latter connects to the resource (http://www.cs.ualberta.ca) and prints all the tags located on the page.

```
Parser parser = new Parser ("http://www.cs.ualberta.ca");
parser.registerScanners(); //register common scanners

for (Enumeration i = parser.elements();i.hasMoreNodes();)
{
            Node node = i.nextNode();
            node.print();
}
```

Figure 9 - An example showing the use of the HTML Parser object.

3.1.2.3 The HTMLReader class

The *HTMLReader* builds on the *BufferedReader* class (from the J2SE IO package) and provides methods to read the data from an input stream. An HTMLReader object is typically initialized with an input reader object (from the J2SE IO package) and the URL to be read. Every parser object has an HTMLReader object associated with it. The HTMLReader's *readElement()* method reads from the input stream one line at a time and invokes a *find()* method on the HTML tag class in order to locate HTMLtags (if any) within the input string. As the HTMLReader reads in lines from the stream in the form of Strings, HTMLNodes are created once HTML tags are located within the input strings.

3.1.2.4 The HTML Tag classes

The tags package contains different tag types that are created mostly by the scanners. It contains a generic HTML tag class, which represents a generic HTML tag. This generic tag class allows the developer to register specific tag scanners, which can identify different tags such as links, image references and others. The generic HTML tag class is extended by different HTML tag classes such as: the HTML Link Tag class, Image Tag class, Title tag class, Form tag class and many others. Each tag class implements a *find()* method invoked by the HTMLReader. This method locates the tag within the input string provided by the HTMLReader, by parsing the string from a given position. A class diagram is provided in Figure 10, showing the relationship between the parent HTMLTag class and its subclasses.

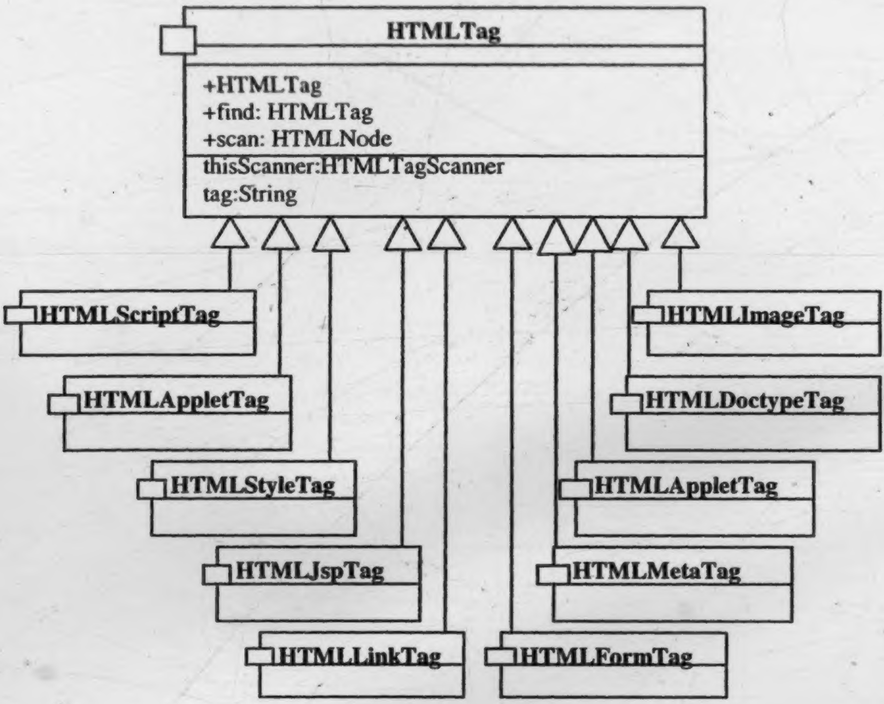

HTMLTag
+HTMLTag
+find: HTMLTag
+scan: HTMLNode
thisScanner:HTMLTagScanner
tag:String
HTMLScriptTag
HTMLAppletTag
HTMLStyleTag
HTMLJspTag
HTMLLinkTag
HTMLImageTag
HTMLDoctypeTag
HTMLAppletTag
HTMLMetaTag
HTMLFormTag

Figure 10 - Class diagram of the HTML tag classes (HTML Parser package)

3.1.2.5 The HTML Scanner classes

The scanners package consists of HTML tag scanners that can be fired automatically upon the identification of the tags. Each scanner is matched to a corresponding tag, forming a scanner-tag pair. For example, an HTML Link Tag class works with the Link scanner class to locate the link tag in a given string. This package contains a generic Tag scanner class, which is sub-classed to create specific scanners that identify the tag, operates on its strings, and can extracts data from it. Each scanner class implements two important methods: and *evaluate()* method and a *scan()* method. The *evaluate()* method is used to decide if the scanner can handle that particular tag type, while the scan() method scans the tag and extracts all the information relevant to it. A class diagram is provided in Figure 11, showing the relationship between the parent HTMLTagScanner class and its subclasses. Figure 12 shows the association between an HTML tag class and its corresponding tag scanner class.

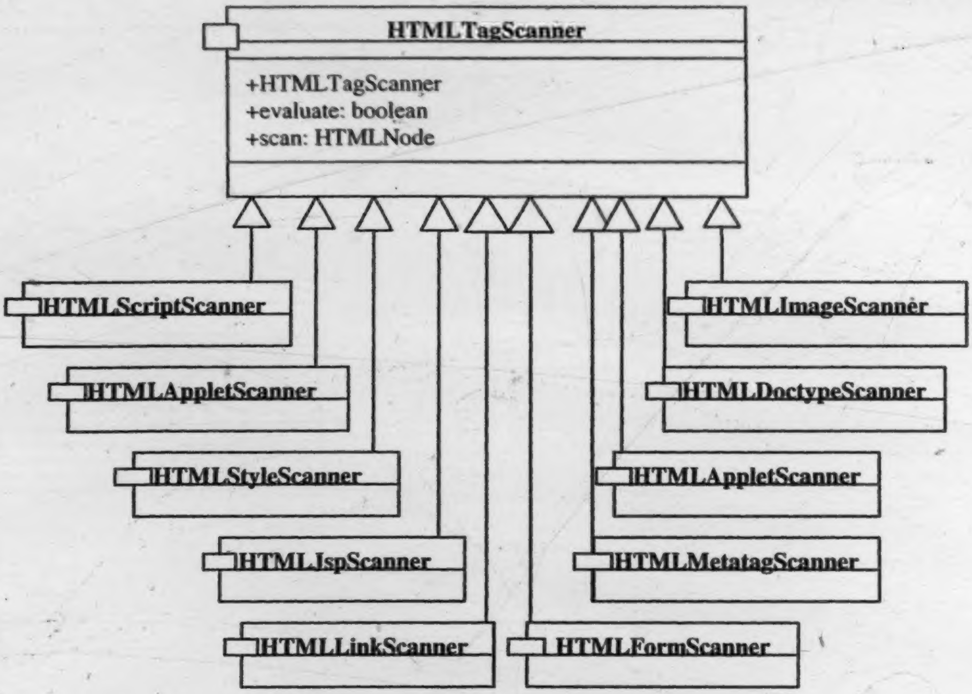

HTMLTagScanner
+HTMLTagScanner
+evaluate: boolean
+scan: HTMLNode
HTMLScriptScanner
HTMLAppletScanner
HTMLStyleScanner
HTMLJspScanner
HTMLLinkScanner
HTMLImageScanner
HTMLDoctypeScanner
HTMLAppletScanner
HTMLMetatagScanner
HTMLFormScanner

Figure 11 - Class diagram of the HTML Tag Scanner classes (HTML Parser package)

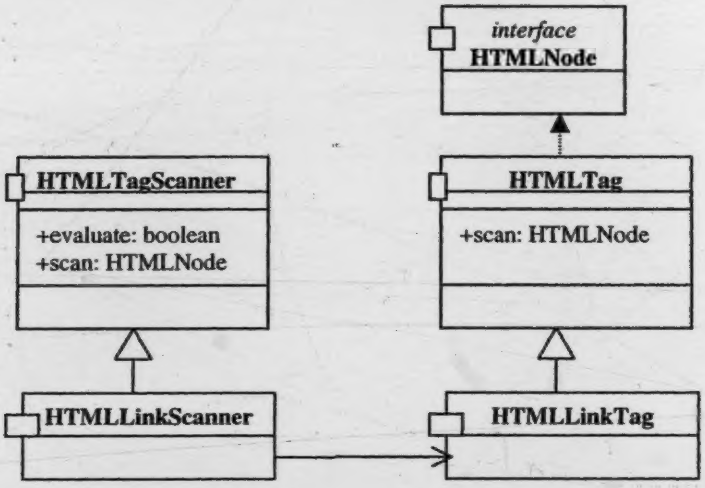

interface
HTMLNode
HTMLTagScanner
+evaluate: boolean
+scan: HTMLNode
HTMLTag
+scan: HTMLNode
HTMLLinkScanner
HTMLLinkTag

Figure 12 - Association between an HTML tag class and its corresponding scanner class.

When the user enters a URL, a connection has to be established to the relevant Web server. An *HTML Parser* object is invoked, which opens a connection to the resource. This HTML Parser object uses an *HTML Reader* object, which reads in all the HTML source code for the requested page and builds *HTMLNode* objects that correspond to the HTML tags located on the page. These HTMLNode objects can be described as ‘capsules’ that hold information pertaining to the HTML tag they represent. For example, consider the following link tag on a web page:
University of Alberta. The HTML link node corresponding to this tag will contain the contents of the tag (*University of Alberta*) and the URL that the link points to *("http://www.ualberta.ca")*. When all the HTML nodes for the page have been constructed, the content is rendered on the screen simply by accessing the information encapsulated in each node object.

3.2 Architecture Implementation

The entire architecture was implemented using the J2ME Wireless Toolkit [JW-J2ME]. This Toolkit provides examples, CLDC/MIDP documentation as well as a customizable environment for emulating the bahaviour of applications on a group of mobile devices. Table 2 below shows emulations of various example devices supported by the J2ME Wireless Toolkit. The devices possess a range of features that are found in mobile devices, all of which support the MIDP specification. The mobile Browser MIDlet ran successfully on all these devices except for the Palm OS device, which uses a different emulator from that provided by the J2ME Wireless Toolkit.

Table 2 - Characteristics of devices emulated using the J2ME Wireless Toolkit

Device	Description	Features
DefaultColorPhone	Generic telephone with a color display (not an approximation of a real phone)	96x128 display resolution, 256 colors, ITU-T keypad with 2 soft buttons
DefaultGreyPhone	Generic telephone with a gray-scale display.	96x128 display resolution, 256 shades of gray, ITU-T keypad, with 2 soft buttons.

MinimumPhone	Generic telephone with minimum display capabilities	96x54 display resolution, black and white color support, ITU-T keypad, no soft buttons
RIMJavaHandheld	RIM device from Research In Motion Ltd.	198x202 resolution, black and white color support, QWERTY-keyboard, no soft buttons
Motorola_i85s	Motorola i85s phone from Motorola, Inc.	111x100 resolution, black and white color support, ITU-T keypad, with 2 soft buttons
PalmOS_Device	Palm OS PDA from Palm, Inc. (this emulation uses the Palm OS Emulator from Palm, Inc._)	Usually 160x160 display resolution, variable black and white color support, Graffiti and hard buttons, no soft buttons

Implementation and Testing using the J2ME Wireless Toolkit Emulator

The J2ME emulator is a tool that enables a programmer to run MIDlets on a desktop computer in order to simulate how the MIDlet will run on a physical device. Even though the goal is to run the MIDlet on the actual device, the emulator plays an important role because it enables the programmer to work entirely on a personal computer throughout the application development process. [Mor01] lists a few of the benefits of using this Wireless Toolkit as follows:

- A MIDlet can be tested not only on one device, but on a variety of different target devices, including custom devices.
- The toolkit provides functionality for monitoring specific aspects of a MIDlet's execution such as class loading, method calls, and garbage collection.
- The emulator serves as a substitute for a physical device during the early stages of MIDlet development when a programmer is likely to make numerous changes in the application.

By testing a MIDlet on the range of devices available within the J2ME emulator, a more accurate approximation of how the MIDlet will function on the real device is achieved, particularly how the application will look when displayed on the mobile device.

3.2.1 Drawbacks of using an emulated environment

Even though the J2ME Emulator displays numerous benefits as described above, there are some setbacks that a programmer must be aware of when using this emulated environment. The emulator does not replace the actual physical device, and the actual performance of an application can only be truly seen when run on the physical device. [Mor01] explains some of the limitations of using the emulator in testing MIDlets:

- The emulator does not properly simulate the varying range of memory constraints that exist between the different devices emulated. This undoubtedly poses a problem when testing MIDlets, since the available memory can drastically affect the performance of the MIDlet.
- More importantly, the emulator provides only an approximation of the physical device. Hence, results acquired from an emulator regarding the execution or performance of a MIDlet is not guaranteed to be identical to that obtained when testing on the actual device.

Nonetheless, the Emulator was an ideal option for this application development stage of this work because it reaped all the benefits provided by its emulated environment.

3.3 Mobile-Resident Browser Evaluation

The baseline MoBed testbed described in the above sections burdens the constrained mobile device with computationally intensive tasks like data fetching, HTML parsing, as well as user interface maintenance. In a more practical scenario, such tasks should be off-loaded to a nearby proxy server to ensure that very little of the limited heap space on the device is consumed, thereby saving time. In this section however, we focus on the scenario where the browser is resident on the mobile, even though the costs are expected to be high.

In order to measure the cost of the computationally intensive activities (such as obtaining the resource, parsing the data, and creating HTMLNodes), a simple experiment was designed to measure the strain on the device in this setting. A simple Driver MIDlet was designed to use the classes in the mobile Browser package to fetch and render pages from data file containing a list of URLs. Figure 13 shows the simple algorithm implemented for the Driver MIDlet.

Driver Test Program

```
- Get fresh heap size
-Read a list of URLs from a text file; FetchAndRender far = new;
 For each URL, U in the list
        far.startApp() //start the fetch and renderer MIDlet.
        Get before_heap snapshot
        far.fetch (U) – fetch U
        Get after_heap snapshot
        far.render (U) – render U
        Get after heap snapshot
        far.destroy()  //clear all use of heap space to start over
```

Figure 13 - Driver MIDlet used to initiate client requests from the mobile client, while gathering information on the heap size change over time

3.3.1 The Dataset

The experiments were conducted using a small set of just over 100 URLs obtained from the server logs from the University of Alberta - Computing Science Department website. This dataset, although small, contained URLs to web pages that were varied in content, structure and size, in order to reflect the natural complexity of typical web pages.

In order to carry out this experiment, the dataset had to be divided into smaller workloads containing a collection of 10 to 12 URLs each. The reason for this was because an attempt to run the experiment with the complete set of over 100 URLs would fail because of the insufficient memory constraint on the device. As such, the experiment was performed using each of the smaller workloads generated from the dataset.

3.3.2 The Experiment

The Driver MIDlet illustrated in Figure 14 above was designed to show the state of a minimum requirement CLDC/MIDP mobile device (with a heap size of 500 kilobytes) as the processes of fetching and rendering the web pages occurred. Figure 14 shows a small snapshot of the output from the test application.

```
Starting Driver ... 482568 bytes
#---------------------------------#
URI = http://www.cs.ualberta.ca/
Before Fetch = 464848 FREE bytes.
Start of FETCH method... 464848 FREE bytes
 Number of HTMLNodes: 389
End of FETCH Method... 209932 USED bytes
After Fetch and Render = 259916 USED bytes.
#---------------------------------#
URI = http://www.cs.ualberta.ca/survey.php
Before Fetch = 411716 FREE bytes.
Start of FETCH method... 411716 FREE bytes
Number of HTMLNodes: 87
End of FETCH Method... 87680 USED bytes
After Fetch and Render = 134032 USED bytes.
#---------------------------------#
URI = http://www.cs.ualberta.ca/contact.php
Before Fetch = 410436 FREE bytes.
Start of FETCH method... 410492 FREE bytes
 Number of HTMLNodes: 255
After Fetch and Render = 97296 USED bytes.
#---------------------------------#
                  ....................
```

Figure 14 - Sample output from the Driver MIDlet

Elaborating on the sample output from Figure 14: the first line shows that the initial heap size for the phone is almost 500 kilobytes (Kb) when the URL *http://www.cs.ualberta.ca/* is requested. After the page is fetched and parsed, the heap snapshot shows that the available memory has drastically reduced to about 209Kb and an additional 50Kb is used up to render the page on the user interface. The number of HTML nodes produced by the parser in this example is 389, showing that this page is reasonably big (as it contains at least 389 parse-able HTML page entities such as images, links and text). Figure 14 also shows that the different pages have a varied number of HTML nodes even though their content is not evident from the snapshot. The number of HTML nodes present in a page only reflects the number of 'parse-able' HTML tags from the HTML Parser program.

3.3.3 Experiment Analysis

Figure 15 below shows the amount of memory (in bytes) used to fetch and render the HTML nodes generated from each of the requested pages, while Figure 16 depicts the amount of time (in milliseconds) used to fetch and render the HTML nodes pertaining to each downloaded page.

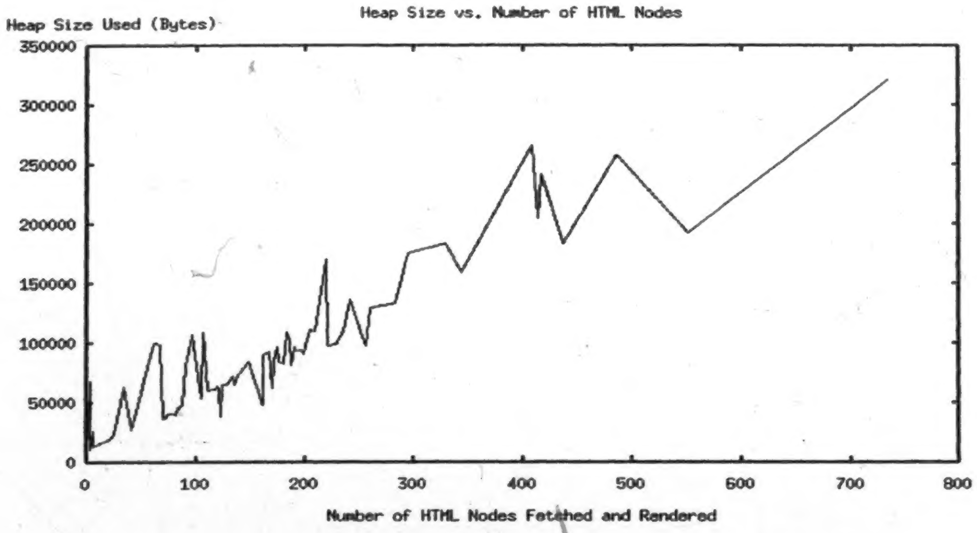


Figure 15 - Required heap size as a function of the number of 'parsable' HTML nodes.

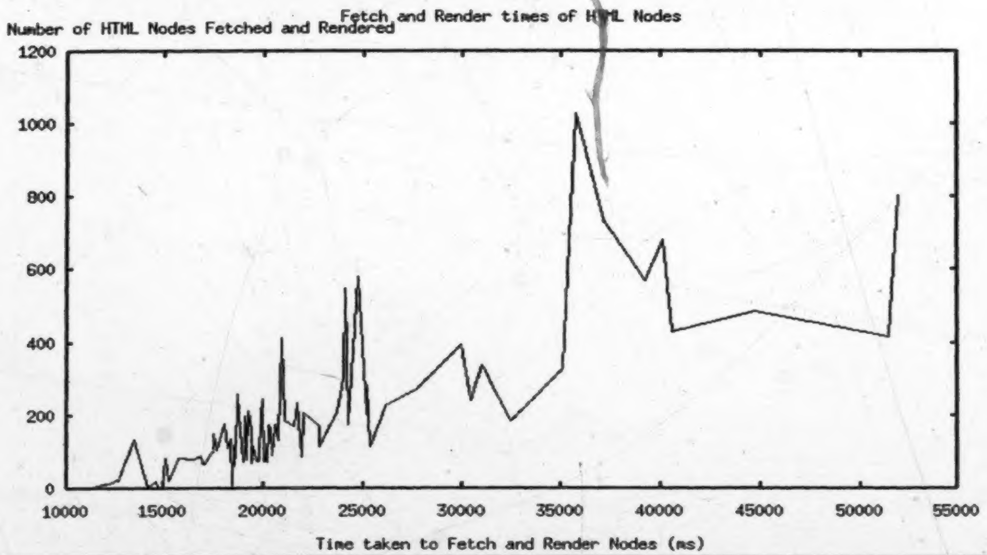


Figure 16 - Time taken to fetch and render a requested page as a function of the number of 'parsable' HTML nodes.

From the above graphs and from monitoring the runtime behavior of the browser application, the following behavior is observed:

- The higher the number of HTML nodes generated for the URL being fetched, the more memory is used to fetch and render the page on the mobile device. This is an expected behavior, since a high number of HTML nodes indicates that the page requested is big and may have a high level of complexity. This behavior is captured in Figure 15 above. As shown by the values in the graph, almost half of the device's entire heap could be used in retrieving a single large web page.
- Figure 16 shows the amount of time used to fetch and render the HTMLNodes. This graph is not as linear as the graph in Figure 15, but there is a general observation that the higher the number of HTML nodes parsed from a web page, the higher the time used up in the fetching and rendering process.
- It was also generally observed that for most of the URLs, more memory was used in the 'fetch' process, and less or equal amounts of memory used up in the render phase. This is because the 'fetch' phase involves: reading data from the connection to the resource, creating HTML node objects and storing them in a vector for rendering. The 'render' phase simply entails iterating through all the HTML nodes stored in the fetch phase, and appending their relevant text information on the mobile client screen object. This clearly explains why more resources are used in downloading the page than in rendering it.
- Although not evident from the graphs above, it is important to mention here that with this HTML parsing scheme, when a page is parsed, HTML nodes are generated for HTML tags that are potentially useless to a mobile client, such as HTML remark tags and end tags. The HTML nodes generated from such tags do not provide any information that will be displayed to the user. Therefore, parsing and storing such nodes on a device with limited disk space is an expensive, and time-consuming task.

3.3.4 Limitations of the Experiment

The results demonstrated above are as expected, but there are a few limitations of the experiment discussed below:

- The experiment provided only an estimate of the cost incurred in fetching and rendering pages at the browser. Most experiments that attempt to measure the size of objects encounter one difficulty: simply taking the difference in memory size before and after an operation proves to be misleading since the JVM may be performing other tasks aside from the operation in question, including allocating and discarding transient objects. As such, it is recommended that the average of the difference (with the number of instantiated object) be taken to reduce the skewing of the results from the background JVM activities [Sin03]. In this experiment, it was difficult to average the memory size difference, since the exact number of objects being instantiated and used was not known. This is because in fetching a resource, several classes in the browser package are invoked, and it is not trivial to know exactly how many objects are involved in the activity. As such, the naïve scheme of simply using the difference in memory size was adopted. Thus, there is no doubt that the results may be skewed, the extent of which is not known.
- The J2ME Wireless toolkit was used in this experiment, and all the drawbacks involved with using the emulator listed in Section 3.2.1 apply here.

The experimental results discussed in Section 3.3.3 clearly show the drawbacks of having a mobile-resident browser, where the mobile device does all the work in getting, parsing and displaying the requested resource. As long as computationally intensive processes are done on-device, the application will not be practical, due to all the constraints of a mobile device. Rather than perfect the application and/or the measurement process of the experiments, this research objective lies with the obvious approach of offloading computation and storage to a nearby proxy process residing on the fixed backbone network. We can tell by the browser example that, even though in absolute values, the number of pages that can be retrieved could be improved, the penalty of having a complete browser implementation on a mobile device is very prohibitive.

Chapter 4 MoBed Client-Proxy Architecture

Web access from mobile devices is characterized by constraints such as small screen and keyboard size, slow connections, and limited bandwidth amongst others. Because of these constraints, small devices need special consideration when accessing information from Web servers. The authors of [CM03] propose an architecture called Scalable Browser for mobile devices, with features such as fetch-on-demand, progressive rendering, and display on demand navigation style. In [BGMP00], the authors introduce five methods for summarizing, browsing and progressively disclosing parts of web pages for small handheld devices. This summarization process is the core of the progressive disclosure mechanism used for mobile clients. [PS01] presents a peer-to-peer data sharing system for mobile users called 7DS, a system that enables data exchange among peers. A small device navigation model for web access called the m-Links [developed by [STHK03], was designed to achieve web navigation on small devices, digging into embedded information on web pages for useful data, separation of service from links, and providing an open framework for others to develop services for wireless. These are only a few cases of past research that show different approaches to making mobile Web access more readily available.

In this architecture, the client resides on the mobile device, while a middleware component or proxy server is used for computationally intensive tasks, such as networking, HTML Parsing and more. In other words, the bulk of the application logic resides on the proxy server while the mobile client is responsible for updating the user interface. The two main components in this architecture are the mobile client and the proxy server. Figure 17 shows a simple sequence diagram illustrating the interaction between the client and proxy components. A detailed description of the client and proxy components is provided in sub-sections 1 and 2, respectively.

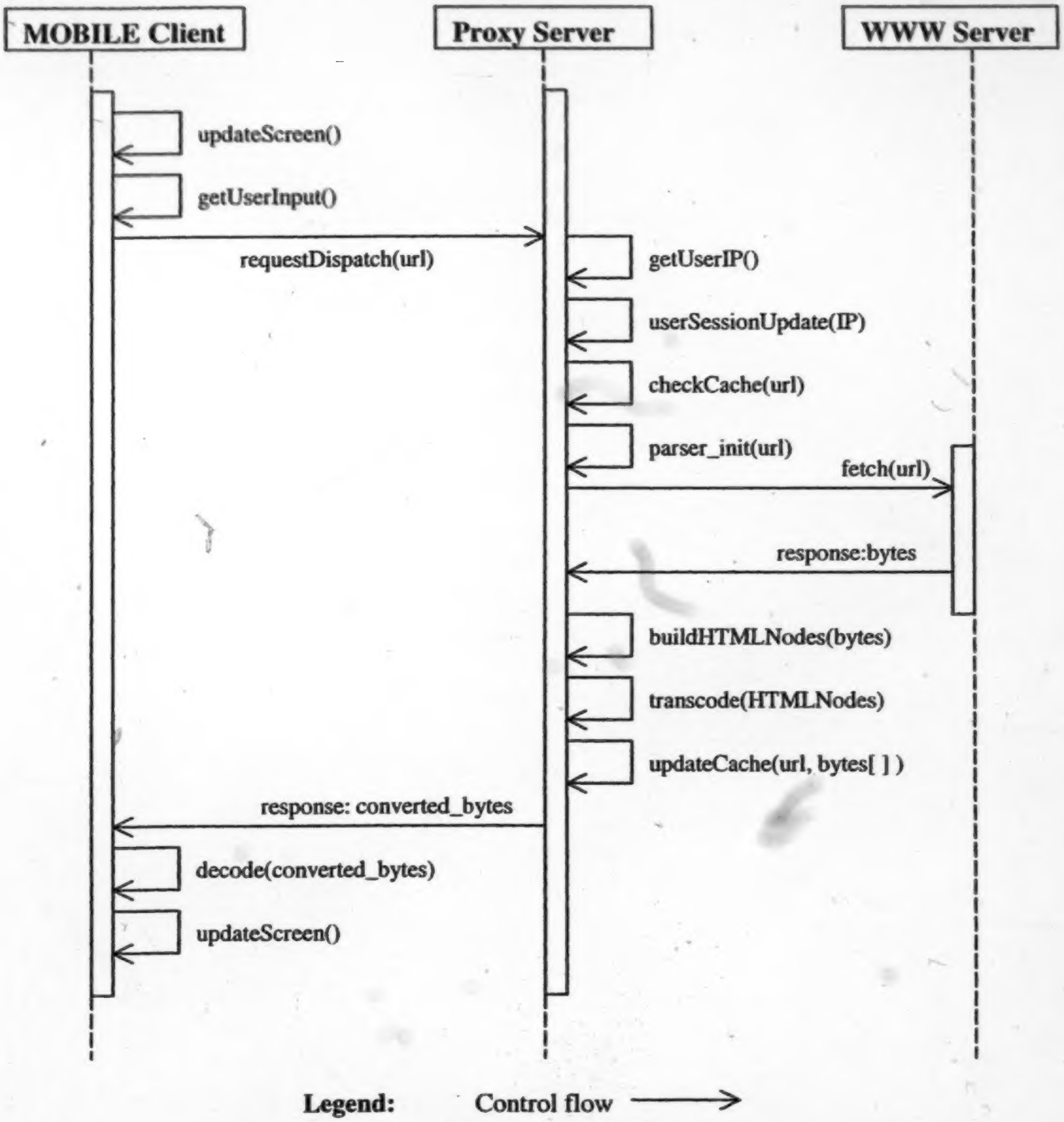

MOBILE Client
Proxy Server
WWW Server
updateScreen()
getUserInput()
requestDispatch(url)
getUserIP()
userSessionUpdate(IP)
checkCache(url)
parser_init(url)
fetch(url)
response:bytes
buildHTMLNodes(bytes)
transcode(HTMLNodes)
updateCache(url, bytes[ ])
response: converted_bytes
decode(converted_bytes)
updateScreen()
Legend: Control flow

Figure 17 - Interaction between main components in the MoBed Client-Proxy architecture

4.1 The Mobile Client Component

As previously mentioned, the client component in this architecture is the J2ME-enabled mobile device. One of the main objectives of introducing a proxy server into this testbed was to remove the bulk of the application logic from the client, in order to achieve a more efficient approach to Web access. Nevertheless, the client still maintained some logic, in addition to its main task of managing the user interface. Figure 18 shows the flow of control and interaction between the main sub-components in the client. Since this sub-section focuses on the description of the Mobile Client Component, the details on the Proxy Server component are provided in sub-section 4.2 (and not illustrated in Figure 9). For each mobile client sub-component described in this section, a brief description of its corresponding component in the Client Baseline Architecture (in Chapter 3) is provided.

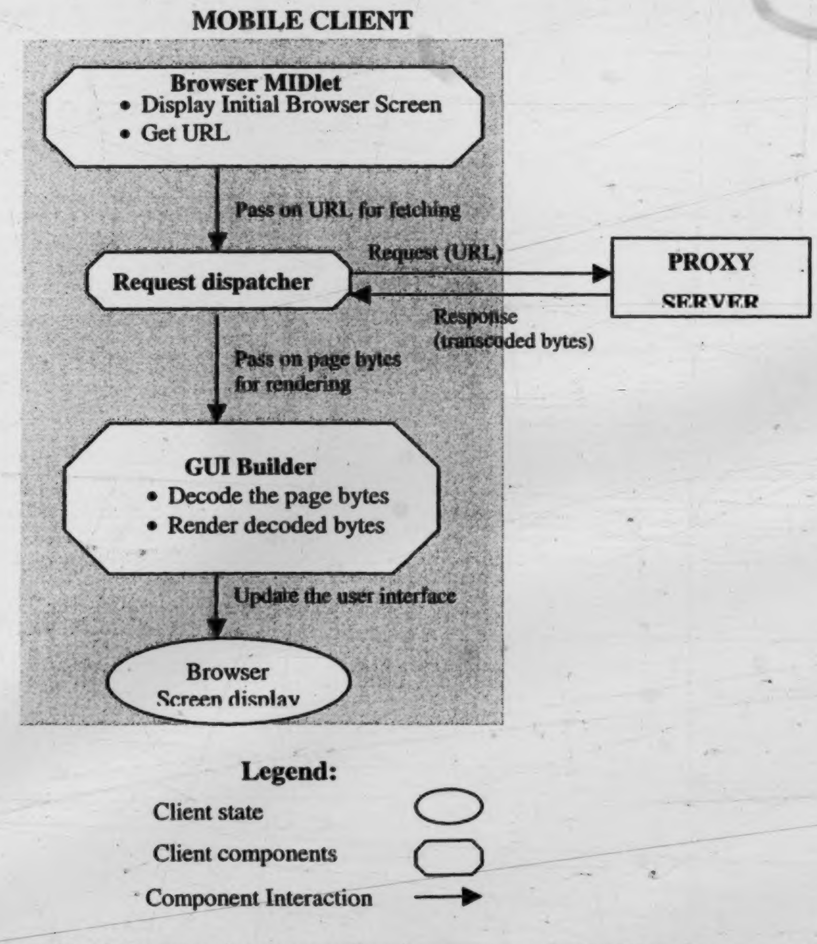

MOBILE CLIENT
Browser MIDlet
• Display Initial Browser Screen
• Get URL
Pass on URL for fetching
Request (URL)
Request dispatcher
PROXY SERVER
Response (transcoded bytes)
Pass on page bytes for rendering
GUI Builder
• Decode the page bytes
• Render decoded bytes
Update the user interface
Browser Screen display
Legend:
Client state
Client components
Component Interaction

Figure 18 – Interaction between main sub-components in the Client Component

The Client consists of the following sub-components:

- The Browser MIDlet, which initiates the browser application.
- The Request Dispatcher that connects to the Proxy when a request is made for a resource.

- The GUI Builder, which builds and updates the user interface based on the bytes received from the Proxy.

4.1.1 The Browser MIDlet

The *Browser MIDlet* has the crucial function of allowing the user access to Web resources. When the application is initiated from the mobile phone, the user interface is updated, giving the user the option to enter a URL. When the URL is received from the user, the MIDlet invokes the Request dispatcher, which connects to the proxy. The MIDlet updates the user interface when the requested page becomes available, allowing the user to view or select the items on the page.

In the Client baseline architecture, a mobile-resident Browser MIDlet was developed, which performed similar functions such as user interface creation and management. The main difference between the two components is that the Browser MIDlet in the baseline architecture, when a URL was retrieved from the client; it invoked the mobile-resident HTML parser with the requested URL. In this testbed, the Browser communicates directly with its connection manager or Request dispatcher and passes on the String parameter for the requested resource. The Request dispatcher then communicates directly with the Proxy server, delivering the user's request.

4.1.2 The Request Dispatcher

The *Request dispatcher* is the connection class that makes the Proxy server accessible to the mobile client. When this component is invoked, it establishes a socket connection to the proxy and sends the URL string request. In future, this class could be modified to send a request in the format: *<String URL, int available space>*; where *URL* is a String representation of the url requested, and *available space* is the integer value of the cache space available for storing any prefetched items from the Proxy. If such a request is dispatched, the client informs the Proxy of its storage constraint for two main reasons. First, to convey its willingness to receive prefetched items; and second, to set a constraint on the amount of prefetched data it can store, thereby saving the Proxy from excessive file prefetching. When a response to the request is received from the Proxy, the dispatcher updates the cache by storing the new page and its content bytes. At this point, the user interface is typically updated to display the new page. In the future, if prefetching to the client is possible, upon receipt of a prefetched item from the Proxy, the cache could be updated with that item, making it available for possible future access.

In the Client baseline architecture, the client establishes a connection directly to the Web server (through the HTML Parser). In this testbed, the implementation is clearly better, as it separates the user interface from the networking functionality.

4.1.3. The GUI Builder

The *GUI Builder* component consists of a collection of classes that perform the very important task of rendering the bytes received for a page, into a displayable format on the user interface. When the Proxy parses a page, it creates special *node*s pertaining to the HTML tags present on the page. These nodes are packaged in a format known to the client, and can be easily unpacked for display. For example, a link tag on a page is converted to a special link node that is easily decoded and displayed as a link on the client user interface.

The GUI builder uses the MIDP User Interface package to create the graphics pertaining to the nodes for each page. A page to be displayed on the screen is essentially a List object on which other graphic items are appended, such as text fields, choice lists, strings and others. When a page is displayed, any images resident in it are not displayed (since the proxy does not download a page's embedded images); images are rendered as links. This way, if a user wishes to view an image, she can click on the link and start downloading the image from the Proxy. Pages that contain forms are similarly approximated – a form is only displayed when a user selects it on the screen.

In the Client baseline architecture, the Browser MIDlet is the only class that creates, and updates the screen when requests are made. This MIDlet extracts the relevant information from HTMLNodes generated from the parsing process, and updates the screen accordingly. In this testbed, when the Request dispatcher receives the packaged, transcoded bytes for a request from the Proxy, its passes the data to the GUI Builder classes which then decode the response bytes and update the screen.

4.1.4 Adding a Cache on the Client

In the future, addition of a client cache to the client could provide an opportunity for the mobile to limit the number of accesses it makes to the Proxy. The Client cache could be implemented as a persistent storage that remains intact when the device is turned on and off. It could store the String representation of a URL, as well as the bytes received from the Proxy that pertain to that page, as shown in Figure 19. A simple cache eviction policy could also be implemented to ensure that the cache is properly managed and occupied with fresh items that reflect the user's ever-changing needs. Depending on the constraints of a particular device, such a cache can be structured to hold more items, as well as implement a stricter or liberal cache eviction policy.

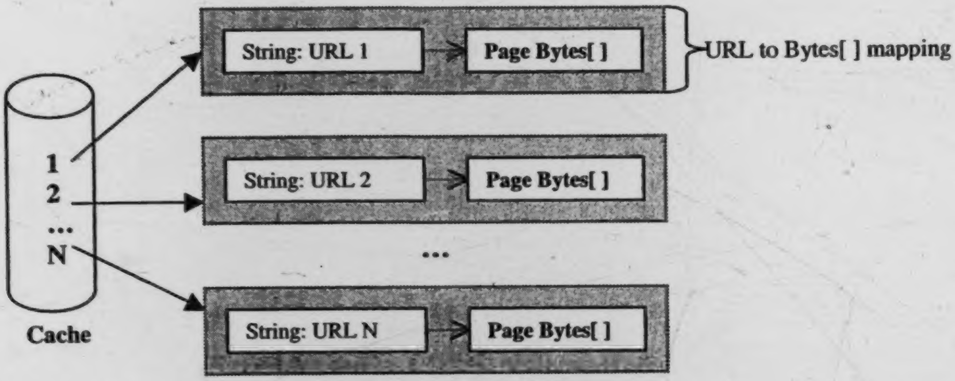


Figure 19 - A view of the Mobile Client Cache

In the Client baseline architecture, there was no persistent client cache component. A collection of already-visited URLs and their content bytes was maintained during a user's Web browsing session. These URLs are lost when the Browser MIDlet is exited.

4.2 The Proxy Server Component

The second major component of this testbed architecture is the Proxy Server. The latter works together with the mobile client to provide access to Web servers. The Proxy server was responsible for the following tasks:

- Receiving, processing and satisfying requests from the client.

- Parsing the HTML content received from the WWW Server.
- Transcoding bytes read in from WWW servers into a 'client-friendly' version.
- Maintaining a Cache, thereby enabling the storage of already visited pages accessed by all clients over time.
- Performing prefetching to clients by using a prediction algorithm to determine what pages(s) a client will likely access next.
- Maintaining a registry of all users that access the proxy, as well as tracking user sessions for the creation of different client profiles.

Figure 20 shows a compact illustration of the Proxy in action, starting from when a client requests a page until it receives a response from the Proxy. The main sub-components in the Proxy are described in the sub-sections that follow.

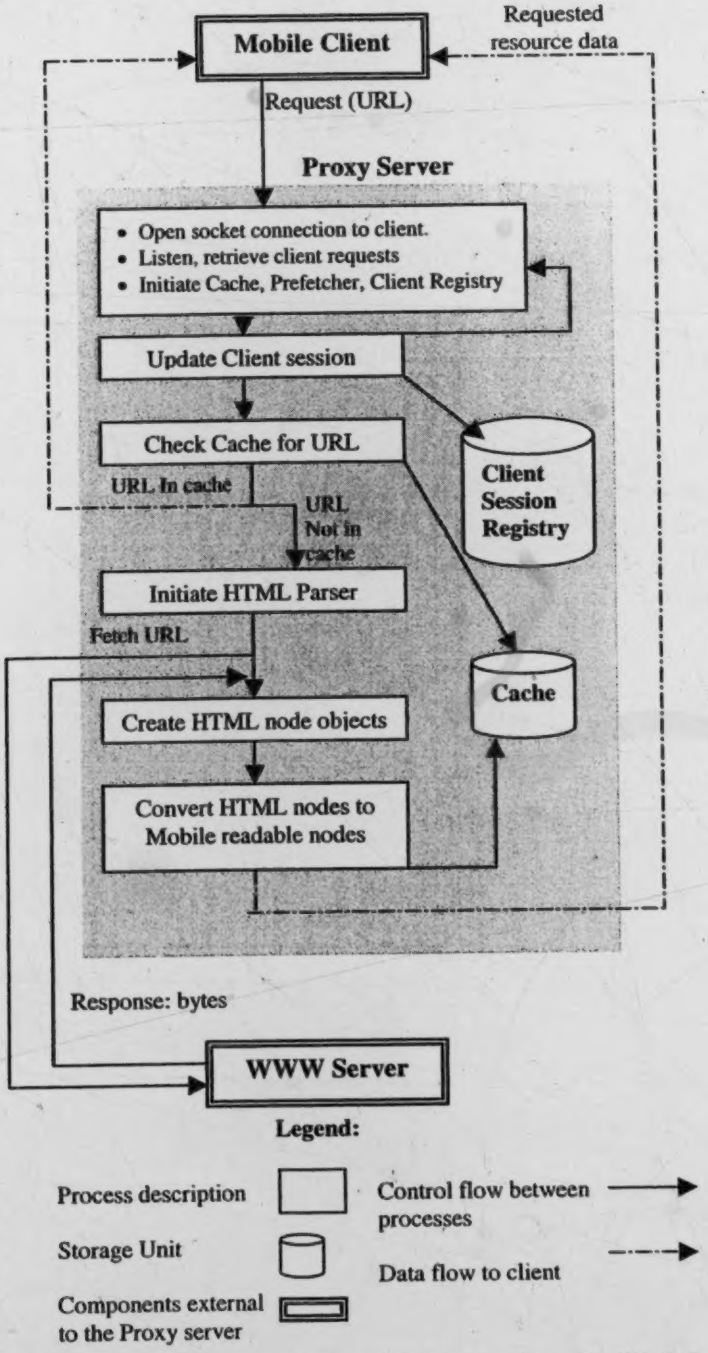

Mobile Client
Requested resource data
Request (URL)
Proxy Server
• Open socket connection to client.
• Listen, retrieve client requests
• Initiate Cache, Prefetcher, Client Registry
Update Client session
Check Cache for URL
URL In cache
URL Not in cache
Client Session Registry
Initiate HTML Parser
Fetch URL
Create HTML node objects
Cache
Convert HTML nodes to Mobile readable nodes
Response: bytes
WWW Server
Legend:
Process description
Control flow between processes
Storage Unit
Data flow to client
Components external to the Proxy server

Figure 20 - The Proxy server functionality using MoBed

4.2.1 The Proxy Controller class

The main controller class on the Proxy provides the connection to the client via a socket connection that accepts a client's requests as they arrive. The mobile client connects to a given port on the wired host machine acting as the Proxy via a socket connection. The Controller class listens to that port and accepts requests for URL sent by clients. Client requests consist of the String representation of the URL requested, and the available space on the client for the storage of prefetched items (which also serves as a prefetch signal).

This class is responsible for initializing the local cache, client session tracker engine, the HTML Parser, and the Prefetch engine. When a request is received, a search for the requested URL is issued on the local proxy cache. If the request is found in the cache, its contents are simply retrieved and sent to the client. If the requested URL is not found in the cache, the Parser object is invoked – beginning the process of making the file available for upload to the client.

4.2.2 The HTML Parser

The same HTML Parser was used in the Client Baseline architecture is used in MoBed as well. For a detailed description of the HTML Parser package, refer to Chapter 3, Section 3.1.2.

4.2.3 The Proxy Transcoder

With the generation of HTML nodes from the HTML Parser (described above), the next step is to convert the nodes into a representation that can be easily unpacked and displayed at the mobile client. When the parser is invoked with the requested URL, it supplies HTML nodes generated from parsed HTML tags from the requested page. As the parser creates HTML node objects, it passes them on to a Transcoder, which extracts relevant tag-specific information from each node, and builds a corresponding basic, no-frills mobile *browser page node (PageNode)*. There is a small package of PageNode classes shown in Figure 21, which reside on the client and the Proxy, ensuring that the client can work with the page nodes when it receives them.

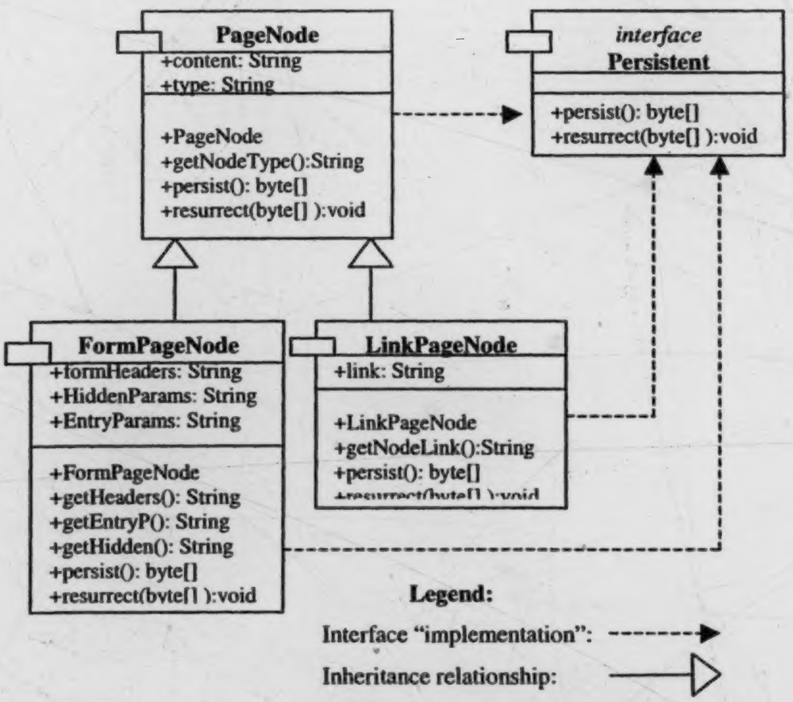

PageNode
+content: String
+type: String
+PageNode
+getNodeType():String
+persist(): byte[]
+resurrect(byte[]):void
interface
Persistent
+persist(): byte[]
+resurrect(byte[]):void
FormPageNode
+formHeaders: String
+HiddenParams: String
+EntryParams: String
+FormPageNode
+getHeaders(): String
+getEntryP(): String
+getHidden(): String
+persist(): byte[]
+resurrect(byte[]):void
LinkPageNode
+link: String
+LinkPageNode
+getNodeLink():String
+persist(): byte[]
+resurrect(byte[]):void
Legend:
Interface "implementation":
Inheritance relationship:

Figure 21 - Mobile Browser Page Nodes

A simple *PageNode* object consists of two main attributes: a *type* string, and a *content* string. Two classes currently extend the PageNode class: the *LinkPageNode* and *FormPageNode* classes. There are currently three main *type* attributes of PageNodes: the "text", "link", and "form" attributes.

- The "text" type represents HTML tags such as title tags, Meta tags, and plain text available on the web page. When any of these HTML nodes is found, the Converter creates a corresponding PageNode of type "text", with its content attribute set to the actual string content of the node.
- The "link" type of a PageNode represents HTML Link and Image tags that both have resource locators pointing to the link or image source. When a link or image tag is encountered, the Converter creates a corresponding *LinkPageNode* object, which has three main attributes: a type string ("link"), a content string (with the textual content of the tag), and a location string (which holds the URL of the image or link).

- The "form" type represents the HTML form tag. A corresponding *FormPageNode* is created when a form tag is encountered. *FormPageNode* consists of its type ("form"), a string for display on the client screen stating that the node is a form, and strings holding the form parameters, and values.

All generated PageNodes are stored in a Vector, which is 'persisted' by writing its contents to a byte array output stream. The data bytes generated from this process are ready to be sent to the client. Persisting the Vector of page nodes is essential since object serialization is not possible using J2ME. When the client receives the data, the vector of PageNodes can be 'resurrected' and its contents accessed. A utility class known to both the client and proxy (obtained from [JTT]) provides the persistence/resurrection capability, ensuring that the client can 'decode' the data when it receives it.

Figure 22 summarizes the data transcoding process on the proxy server.

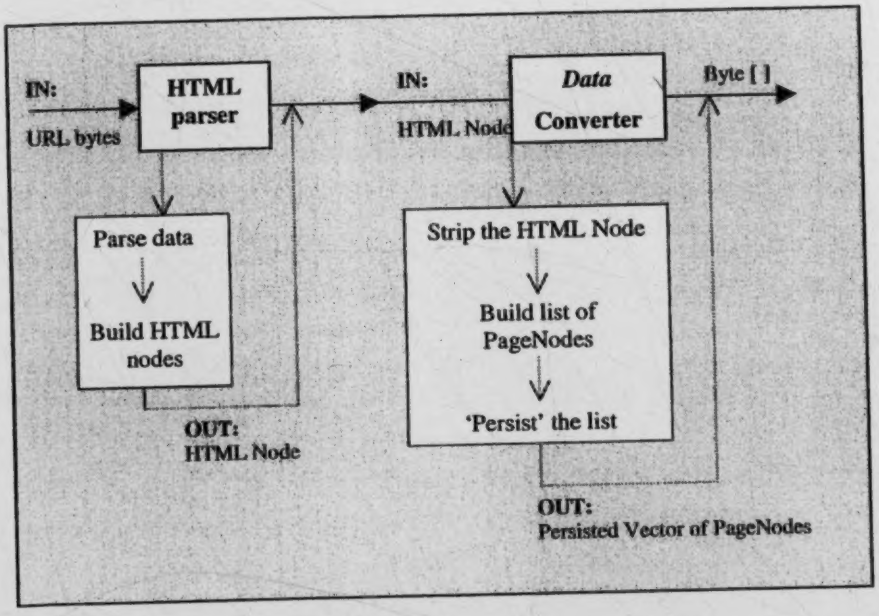

IN:
URL bytes
HTML parser
Parse data
Build HTML nodes
OUT:
HTML Node
IN:
HTML Node
Data Converter
Byte [ ]
Strip the HTML Node
Build list of PageNodes
'Persist' the list
OUT:
Persisted Vector of PageNodes

Figure 22 – The MoBed Proxy Transcoder functionality

4.2.3.1 The Proxy Cache

The Proxy maintains a local cache, which contains all the pages that have been downloaded from Web servers, upon the requests of clients. Having a cache on the Proxy is beneficial in many ways:

1) The cache holds all previously visited pages requested by all clients, hence eliminating the need to fetch those pages again from the Web server when they are later requested (this is true for static pages).
2) The proxy cache is a collection of the URL requests of all the clients that access that proxy. When a page is downloaded to the proxy, if it is not already in the cache, the cache is updated with this new file (regardless of which user requested it). This ensures that other users who request the same page at a later time will potentially benefit from the initial page download.
3) To summarize the two points above, a local cache on the Proxy demonstrates the potential for bringing the contents of a Web Server closer to the user, thereby reducing the observed latency or delay when a page is requested. This also reduces the load on the network and the server.

In the MoBed proxy server, the cache is maintained as a Hash Table, consisting of URLs mapped to their corresponding page bytes. The bytes stored are parsed, transformed HTML nodes from the page, in a format ready to be sent to the mobile client. When a page is added to the cache, the string representation of its URL is hashed to a unique string using the MD5 hashing scheme provided by [MD503]. The hash string is then mapped to an array containing the page bytes, as shown in Figure 23.

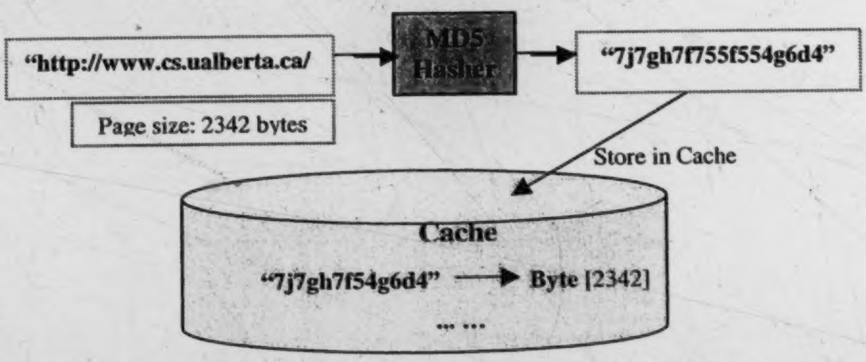


Figure 23 - An example illustrating the addition of an element to the proxy cache

4.2.3.2 Cache management

To cope with the limited resources of a caching proxy server, cache eviction schemes can be implemented to determine what document(s) should be replaced when the cache is full and a request arrives for a URL that is not in the cache. Different cache eviction policies evict documents from the cache based on various cached document attributes, such as: the latest access time of a page, the size of a page, the frequency at which a page is accessed, etc. When cache replacement schemes are not implemented, a proxy can maintain an infinite cache so that the proxy contains all documents ever accessed at any given time. In MoBed, the following two cache replacement schemes were investigated on the proxy cache. All the details on this investigation are described in Chapter 4. Suppose the proxy receives a request for a URL, U that is not in the cache.

- *Least Recently Used (LRU)*

In this scheme, the least recently used document is discarded and replaced with U. When implementing this policy, all cached documents were sorted by their access times, such that newly cached documents were the most recent ones, located at the top of the list. Whenever a document was accessed from the cache, its access time was updated and the document moved to the top of the list. When a request arrives for a URL that is not in the cache, and the cache is full, the document at the bottom of the list is evicted, and the new document stored at the top.

- *Least Frequently Used (LFU)*

In this policy, the least frequently used document is discarded and replaced with U. All cached documents were sorted by their access frequency counts, allowing cached documents with the highest frequencies to be located at the top of the list. When a document was accessed from the cache, its frequency count was incremented and the document moved to the top of the list. When a request arrives for a URL that is not in the cache, and the cache is full, the document at the bottom of the list is evicted, and the new document stored the top.

4.2.4 Session Tracker Engine

Given that many clients access the proxy server at different times, requesting various documents, the Proxy has to maintain some information about the clients it services such as: maintaining a registry of all clients and keeping track of all user sessions over a given time period. This information can also be used as the basis for prefetching mechanisms based on the access history of a set of clients.

The MoBed Proxy tracks all user sessions over a 30 minute period and uses these user profiles to predict possible documents that can be accessed by the client, and subsequently prefetching these documents to clients who are interested in receiving such files. The following sub-sections provide a detailed explanation of the prediction algorithm implemented on the MoBed proxy server; and illustrates the importance of tracking sessions and maintaining user profiles.

4.2.4.1 The Prediction Engine

One of the goals of this project was to determine how prefetching can improve Web access for limited mobile clients using MoBed. The goal is to reduce the latency or delay perceived by a client when a request for a page is made. Caching proxies are known to reduce latency to a fixed amount, but there is a limit to the extent of benefits reaped from caching. A caching proxy has the advantage of bringing Web content closer to the user by storing requests over a wide range of clients who may potentially request files already cached from other users' previous requests.

A prefetching proxy goes a step further - it predicts a user's next request, fetches the content, and sends it to the user before the page is requested. This raises a general concern that prefetching may lead to unnecessary increase in network traffic. However, assuming the proxy performs prefetching when it is idling with no client requests to process, there is a probability, P that the latency of future client requests will be reduced; where P is the probability of correct guesses (of requests) prefetched to a client.

A prefetching and caching proxy takes this idea another step further - not only does it store previously visited URL requests, but it also pushes documents to clients possibly from its local cache, further reducing the possibility of high client-perceived latency when a request is made in the future. The goal is to take advantage of proxy idling between requests to push documents to a client. The MoBed proxy maintains a local cache, and implements a prediction algorithm (discussed in following sub-section) that generates informed guesses of future client requests.

4.2.4.2 Prediction using Path Profiles

The technique used to predict URLs on the MoBed proxy is based on the prediction algorithm originally coined by Schechter et al. [SKS98] which uses path profiles generated from past user requests. The authors describe an algorithm for efficiently generating path profiles from information contained in standard HTTP server logs. The key terms used in describing this prediction algorithm (as defined by [SKS98]) are outlined below and illustrated in Figure 24:

- A *path* is defined as a sequence of URLs accessed by a single user, ordered by the time of access. A path may contain repeated instances of a request; and the length of a path is the number of URL requests that make up the path.
- A *user session* is the path that describes the full set of requests (ordered according to time of access) from a particular user within a specific time frame.
- A *Path profile* is a set of pairs, each of which consists [illegible] a path and the number of times that path occurs over a given time period. A profile is recorded over the set of all user sessions.

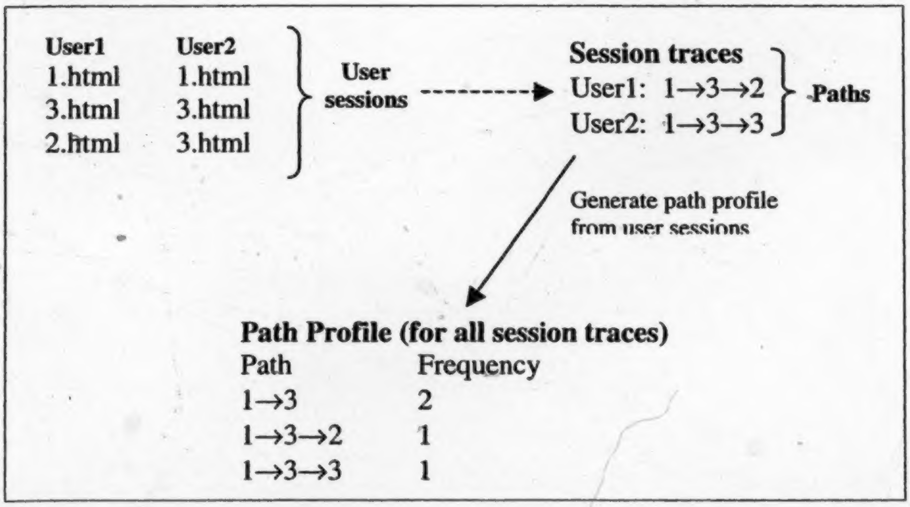

User1
1.html
3.html
2.html
User2
1.html
3.html
3.html
User sessions
Session traces
User1: 1→3→2
User2: 1→3→3
Paths
Generate path profile from user sessions
Path Profile (for all session traces)
Path
Frequency
1→3
2
1→3→2
1
1→3→3
1

Figure 24 - An example showing the relationship between user sessions and path profiles

4.2.4.3 Generating Path profiles

[SKS98] describes two main ways of collecting path profiles: by using an HTTP client that records user paths or by using an HTTP Server that records the paths of all users that access its site. Standard server logs can be used to generate profiles from an HTTP server that contains a large number of accesses. [SKS98] describes an algorithm for efficiently generating path profiles from information contained in standard HTTP server logs. The main points on server-side profiling from the authors are outlined below:

- Most HTTP logs have five main fields that describe each request: the date, time of access, Source (Client) IP Address, the name of the requested file, and the parameter field (derived from the URL). When generating path profiles, a decision has to be reached about the URLs used to form paths: whether they should contain the parameter field or only the name of the file or script. Finding an automatic method for determining which parameters to ignore and which should be considered part of a URL remains an open problem [SKS98]. In this project, all profiles generated do not include parameters as part of the URL.
- The URLs used for profiling are compressed down to a compact format by mapping a unique integer to each unique URL in the log.
- The concept of a user is essential to path profiling because predictions are made to a current user based on the access history of other users who behaved similarly to the current user; thereby showing the importance of differentiating between all users. The IP addresses for each user is present in most server logs, and is used as the identifier for each client (even though this IP could actually represent a proxy server).
- When creating user sessions, all HTTP requests separated by more than thirty minutes are not considered to be part of the same session. This heuristic has to be used to handle cases where users are browsing pages on another site, in between accesses on the server.

4.2.4.4 Path Tree Construction from user sessions

Upon generation of user session paths from HTTP server logs, the sessions are used to generate a tree of important paths [SKS98]:

A path tree begins with a root node and contains nodes that may have a varying number of children. Walking from the root node down the tree is equivalent to walking through a path of URLs. When recording a path in the tree, the first URL in a path is stored as a child of the root node of the tree; the second URL in the path is stored as a child node of the first URL's node. This may continue until the end of the path.

In order to prevent the tree from growing too large, [SKS98] introduces the concept of a *maximal prefix* of a path. The *maximal prefix,* M_p of a path P contains the sequence of URLs in P except for the last one. A path is recorded in the tree if the maximal prefix of that path has occurred at least T times, where T is a threshold that can be configured based on available memory resources.

Each user session is represented as a collection of integers (corresponding to URLs), stored in an array object, which occur in the same order as the URLs in the user session. When the algorithm is initialized, the *PathTree* consists only of a root node. Each tree node, except the root, is labeled with a URL number. When a node is created, it is assigned an *OccurrenceCount* value of 0, except for the root, which is initialized with an *OccurrenceCount* of T. The complete PathTree is constructed by applying the algorithm in Figure 25 to each sequence of URLs that represent a user's session. Figure 26 shows an example of a path tree for storing path profiles.

After the first iteration of the algorithm, the *OccurrenceCount* variables of all tree nodes are zeroed (except for the root). The algorithm is re-ran over the set of all user sessions and the shape of the PathTree is refined. The algorithm must be re-iterated to generate accurate counts of all paths with immediate predecessors that occur at least T times. Before each supplemental iteration of the algorithm, the OccurrenceCount values of all leaf nodes are cleared so that the new-leaf threshold is not reached prematurely. In order to make the final counts accurately reflect path frequencies, a final iteration of the algorithm is performed with the tree structure in place but with all OccurrenceCount values reset to zero [SKS98].

```
• FOR each URL in the sequence (stepping through using a Counter)
    o CurrentNode ← root node of PathTree
    o Index ← Counter
    o DO
        ▪ Increment the OccurrenceCount of the CurrentNode
          (CurrentNode->OccurrenceCount)
        ▪ URL_Number ← URLSequence[Index]
        ▪ IF there does not exist a child of CurrentNode labeled with
          URL_Number AND CurrentNode->OccurrenceCount >= T
            ▪ Create a child node of CurrentNode labeled with
              URL_Number
        ▪ IF there exists a child of CurrentNode labeled with
          URL_Number
            ▪ CurrentNode ← Child of CurrentNode labeled with
              URL_Number
        ▪ ELSE
            ▪ EXIT Do/While Loop
    o WHILE (++Index < length of URL_Sequence)
• END - FOR
```

Figure 25 - PathTree construction algorithm that accepts a list of URLSequences (Algorithm extracted from [SKS98]).

Session traces
User1: 1→3→2
User2: 1→3→5
User3: 3→5

Path Profile
(For all session traces)

Path	Frequency
1→3	2
1→3→2	1
1→3→5	1
3→5	1

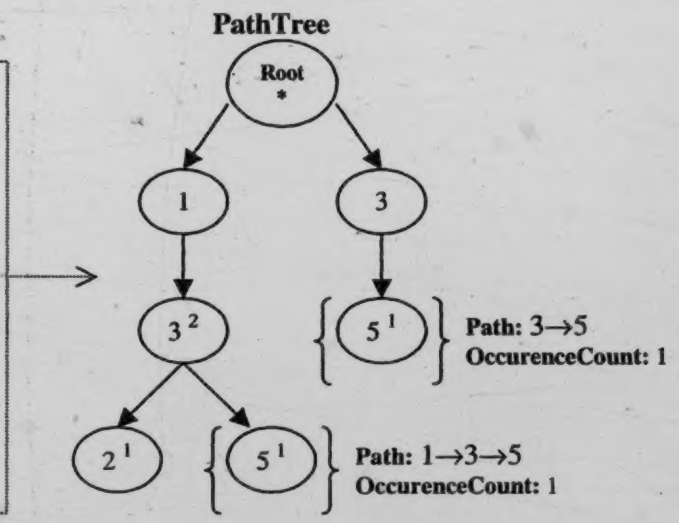


Figure 26 - Example showing how paths are maintained in a PathTree as path profiles.

4.2.4.5 Prediction using the Path Tree

When the path tree is constructed as described above, it is used to make a prediction for the next URL. In order to make the prediction algorithm more efficient, the path tree can be transformed into a more condensed representation of path profiles. The motivation for this is to eliminate the paths not needed for prediction, thereby facilitating the search for paths that match a user's access history:

4.2.4.5.1 Condensation of Path profiles

In order to condense the path profiles, the following steps are executed (and illustrated in Figure 27). A list of all the paths in the tree is constructed by iterating through the path tree. Each path is then separated from its frequency count. The most recently accessed URL in a path is the most important predictor because the page returned by that URL contains the hypertext links from which the user is likely to choose his next destination [SKS98]. After separating the paths from their frequency counts, the last URL is separated from the rest of the history path and becomes the prediction for that path.

The list of all paths are then stored in reverse order, with each entry representing a reverse-ordered path and the number of times that the path occurred. Longer paths that make the same prediction as their shorter counterparts could then be filtered out [SKS98]. The final step is to sort all entries by the reversed history path. If two entries have the same path, the one with the smaller frequency count is eliminated. Predictions are then made using these condensed path profiles. Each entry in the condensed path list is made up of three main elements: the reversed history path, the prediction (the last URL extracted from the original history path) and the frequency count (the number of times that path occurs).

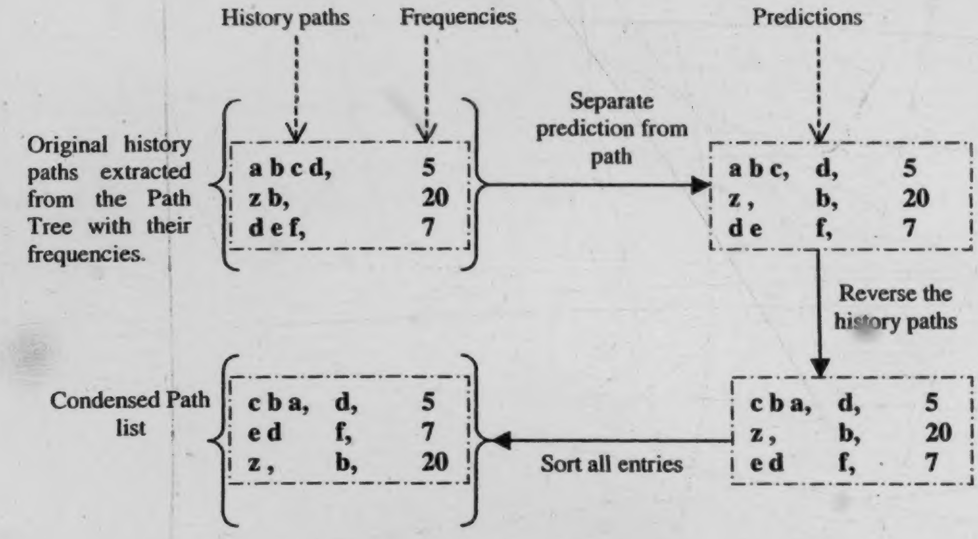

History paths
Frequencies
Predictions
Original history paths extracted from the Path Tree with their frequencies.
a b c d, 5
z b, 20
d e f, 7
Separate prediction from path
a b c, d, 5
z , b, 20
d e f, 7
Reverse the history paths
c b a, d, 5
z , b, 20
e d f, 7
Sort all entries
Condensed Path list
c b a, d, 5
e d f, 7
z , b, 20

Figure 27 - Condensing path profiles

4.2.4.5.2 Prediction from the Condensed profiles

To predict the next URL that a user will request, the algorithm proceeds by first obtaining the user's current session trace. In the MoBed Proxy, the session tracker engine provides the current session for a client when provided with its IP address. Recall that a client's session consists of all the URL requests from that user within a thirty-minute time frame. The user's session trace is then reversed. The condensed path list is then searched through, to find the path in the profile that matches the most consecutive characters in the user's reversed session trace [SKS98]. The chosen URL for prefetching is the prediction element of the list entry chosen from the profile.

The MoBed proxy takes this one step further by making a prediction only when there already is a path in the profile that matches the user's reversed session trace exactly. Even though this cuts down on the number of proxy predictions, this conservative approach to prediction is adopted because the MoBed proxy services mobile clients that have limited storage capacities, as well as bandwidth. This stresses the importance of pushing documents to clients only when the proxy has enough information to make the prediction. Since the condensed path list is already sorted by the reversed history paths, the best path chosen for prediction can be found using a binary search method, bound by *O(log2(number of paths))*.

Figure 28 uses an example to illustrate how the condensed path list is used for making predictions on a user's current session.

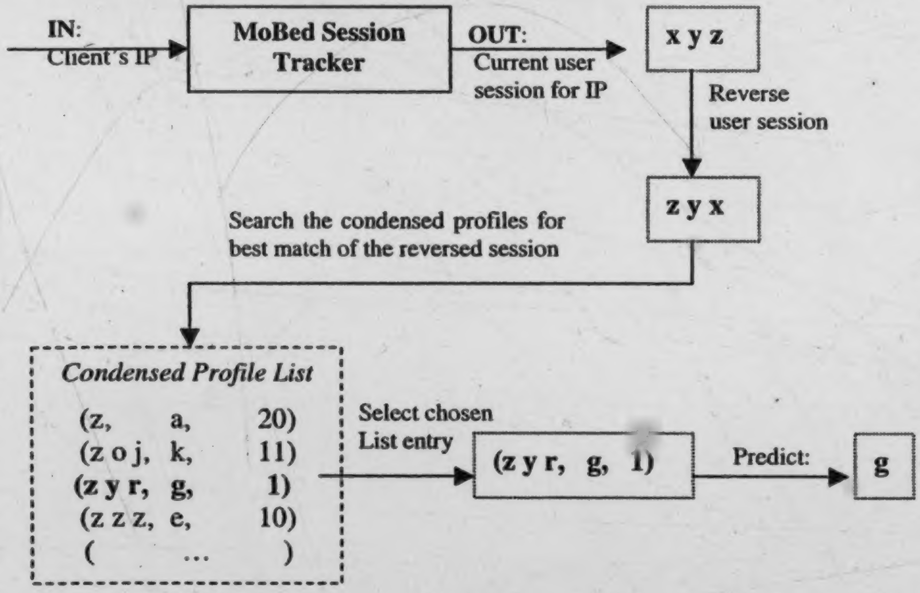

IN:
Client's IP
MoBed Session Tracker
OUT:
Current user session for IP
x y z
Reverse user session
z y x
Search the condensed profiles for best match of the reversed session
Condensed Profile List
(z, a, 20)
(z o j, k, 11)
(z y r, g, 1)
(z z z, e, 10)
(...)
Select chosen List entry
(z y r, g, 1)
Predict:
g

Figure 28 - Predicting using Condensing path profiles

When the proxy receives a request from a client, and the client desires documents to be pushed to it, the MoBed proxy's prediction engine (which runs the path profiling scheme described above), uses the clients current user session to predict what URL it may request next. The predictability of requests is measured using training and testing data sets, for constructing the path list and testing the prediction engine (respectively). This experimental setup is extensively described in Chapter 5.

Chapter 5 Empirical Evaluation

As previously described, MoBed provides an experimental test-bed for designing, developing and analyzing different caching and prefetching schemes that can be used in devising Web access solutions for J2ME-enabled devices. In this testbed, the client is resident on the mobile device, while the proxy server resides on a wired host between the mobile client and the web servers. This architecture allows for various web-access functionalities to be flexibly distributed between the client, proxy and server; resulting in numerous possible configurations for experimentation. Using MoBed, this thesis investigates the following scenarios for web access:

1) Location of a local cache (at the client or proxy server or both);
2) Caching using two main Cache replacement algorithms;
3) Prefetching data from the proxy to the client, using user access history analysis.

Keeping the above points in mind, a suite of experiments were designed to address each of the listed scenarios. Details on the design, results and evaluation of all the experiments are provided in the upcoming sub-sections, with a summary provided in Table 3 below.

Table 3 - Listing of Experiments performed using MoBed

Experiment		Factor(s)	Response variable(s)	# of Simulation RUNS
1		- Proxy location - Caching scheme - Proxy cache size	Proxy-to-Mobile response time	8
2		Original number of data bytes from Web Server	Number of Proxy-Transcoded data bytes for client	N/A
3	3-1	Client cache size	- % of Proxy accesses; - % of Prefetch-interrupts from large predicted files; - Number of files prefetched to clients; - Number of clients prefetched to.	51
	3-2	- PathTree size (determined by T-value) - Using a Retraining phase - Workload size	- % of Predicted File Hit Ratios - % of Predicted Byte Hit Ratios	

A simulation study was chosen as the best way to study the performance of the testbed at this time for two main reasons. Firstly, workloads obtained from server logs can be used as input to a simulation. Such server logs are readily available from web site traffic traces and provide URLs that can be used to simulate client requests. Secondly, a simulation can help in identifying the effect of a number of different factors on the performance of the proxy in delivering web content to the client. Such factors include: the size of the local proxy cache, the cache replacement policies used, and the accuracy of the proxy prediction scheme.

It is important to mention here that the workloads used for all the MoBed experiments were not used for the baseline architecture. Recall from Section 3.3.1 that the dataset used for the Baseline architecture evaluation had to be divided into smaller workloads containing a collection of 10 to 12 URLs each. This dataset partitioning was essential because an attempt to run the experiment with the complete set of over 100 URLs failed because of the insufficient memory constraint on the device (which was a minimum requirement phone with a heap size of 500 kilobytes). For the Baseline architecture evaluation, using the large workloads utilized in the MoBed experiments would be extremely time-consuming since these workloads are considerably large, with thousands of requests. For this reason, a smaller workload was chosen for the Baseline architecture evaluation

In describing the experiments carried out in this chapter, the following standard terminology was used: A *factor* is an independent variable that affects the outcome of a desired response. A *response variable* is a dependent variable that is affected by manipulating an independent variable. An experiment *level* represents a value taken by a factor. A *nested* experiment is one in which the levels of one factor are chosen as a function of the levels of another factor. A *complete factorial* experiment is one in which all of the possible combinations of levels for each factor are investigated.

5.1 Experiment 1: Caching restricted to the Proxy level

The goal of this experiment was to investigate the benefit of introducing caching only at the MoBed proxy.

5.1.1 Objective

This experiment assessed the latency incurred at the proxy when processing a request, i.e. the time taken to fetch, parse, and transcode web content at the Proxy before responding to the client

5.1.2 Workload Description

The workload used for this experiment consisted of server traces collected from the CMPUT 301 course website at the University of Alberta in the Fall of 2001. The traces were made up of requests generated from repeated accesses to 116 distinct URLs from the course site at different times; resulting in 14,000 URL requests and their corresponding timestamps.

This workload was used to simulate actual URL requests issued from a J2ME client to the proxy. A simple J2ME program was designed to run on a mobile client using the J2ME Emulator. The client iterated through the dataset and fired off requests to the proxy. Each request consisted of the URL string and the original request timestamp retrieved from the trace. The client only made another request after it received the data content for its previous request from the proxy. The latter received URL requests and processed them while collecting statistics on the time taken to process each request.

5.1.3 Experiment Design

In this experimental setting, the client was 'passive' i.e. it only fired off URL requests to the client and did not maintain a cache or perform any other functions apart from user interface maintenance. There are three interesting factors in this experiment:

1. The physical location of the proxy server

The location of the proxy server with regards to the client can greatly affect the latency observed after a user requests a URL. In the first scenario, the proxy is *remote*, as shown in Figure 29, located on a separate machine from the WWW and physically closer to the client. In the second scenario, the proxy is located on the WWW server machine (illustrated in Figure 30).

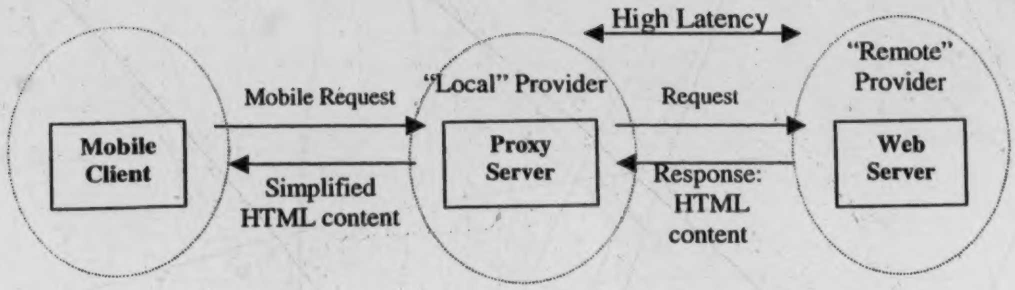

High Latency
"Remote" Provider
Mobile Request
"Local" Provider
Request
Mobile Client
Proxy Server
Web Server
Simplified HTML content
Response: HTML content

Figure 29 - Experiment 1- Proxy location factor (Level 1): Remote proxy server

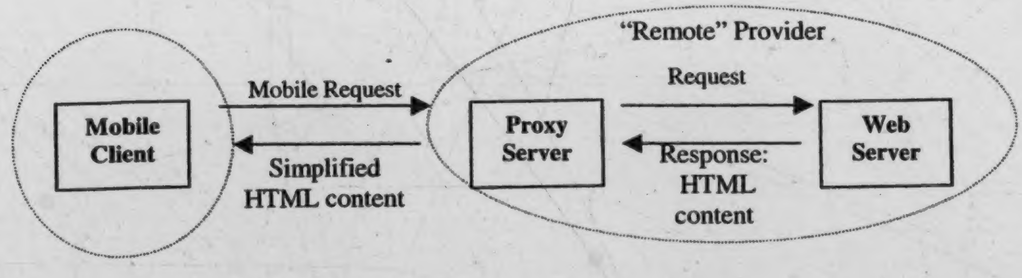

"Remote" Provider
Request
Mobile Request
Mobile Client
Proxy Server
Web Server
Simplified HTML content
Response: HTML content

Figure 30 - Experiment 1- Proxy location factor (Level 2): Proxy located on the Web server

Recall that the goal of this experiment is to measure the latency incurred at the Proxy server when a request is handled. As such, the observed proxy-to-server latency is of primary importance, as it would affect the overall processing time of requests at the Proxy. In Figure 29, the proxy server is located on a separate machine from the Web server, while in Figure 30, the proxy physically resides on the same machine as the Web server. The proxy-to-server latency in Figure 29 is expected to be higher as compared to that in Figure 30 because of the physical closeness of the proxy to the Web content in the latter scenario. As such, the two levels for this factor (location of the proxy) are: *Remote* and on *Web server*.

2. *The basis of caching with different cache replacement schemes*

Two well-known cache eviction schemes were implemented and tested separately at the proxy: the Least Recently Used (LRU) and Least Frequently Used (LFU) policies. The description of these two schemes is provided in Section 4.2.3.2. The two levels of this factor are: *LRU* and *LFU*.

3. *The size of the Proxy cache*

The proxy cache was designed to store *Cacheable* objects. A *Cacheable* object consists of the URL string for a given request, and its transcoded content (a simplified version of the HTML content for the mobile client) shown is Section 4.2.3.1. At the start of an experiment run, the cache storage capacity was set to a fixed number, representing the number of *Cacheables* that can be stored in the cache before any eviction. There are two main levels to this factor (size of the Proxy cache). The caching performance was investigated in this experiment using two different sizes: 50 and 100. Recall from Section 5.1.2 above, that the workload used in this experiment consisted of 116 unique URLs. In order to study the performance of the different cache replacement schemes, the selected cache size levels had to be less than the total number of URLs present in the workload; to ensure that the cache attained its maximum capacity during every simulation run. With a cache size level of 100, the proxy cache was almost infinite (since the unique number of requested URLs was only 116); and with a cache size level of 50, less than half of the unique URL requests would be maintained in the cache at a time. This variation in the cache size provided an interesting heuristic for evaluating the performance.

This experiment consisted of a 3-factor (Proxy location, Caching policy, Cache size), complete factorial experiment, where each factor had two levels; requiring a total of 2^3 (eight) experiment runs. The response variable in this experiment was the Proxy-to-Mobile response time incurred after a request is made. This experiment design is summarized in Table 4 below, showing the different factors and levels for each run.

Table 4 - Factor-level combinations for Experiment 1

Run	Factors		
	Proxy Location	Caching Policy	Cache Size
1	Remote	LRU	50
2	Remote	LRU	100
3	Web server	LRU	50
4	Web server	LRU	100
5	Remote	LFU	50
6	Remote	LFU	100
7	Web server	LFU	50
8	Web server	LFU	100

5.1.4 Results and Evaluation

The time taken to process the web content at the Proxy was recorded for each cache eviction policy, each cache size (50 or 100 cacheable objects) and for each proxy location scenario (remote or on the Web server); resulting in eight experimental runs (shown in Table 4).

Table 5 - Observed Proxy latency for all simulation runs in Experiment 1

Percentage of Proxy processing latencies within the given time range						
RUNS	**Description**	**(< 1 ms)**	**(1 to 10 ms)**	**(<= 500ms)**	**(> 500ms)**	**Average latency**
1	LRU_Remote_50	14.5	67.9	99.8	0.2	23.05
2	LRU_Remote_100	3	77.9	99.7	0.3	54.34
3	LRU_WebServer_50	0	96.2	99.9	0.1	10.03
4	LRU_WebServer_100	0	93.5	99.9	0.1	11.6
5	LFU_Remote_50	5.5	73.8	99.8	0.2	22.17
6	LFU_Remote_100	11.5	69.7	99.8	0.2	22.78
7	LFU_WebServer_50	0	91.9	99.8	0.2	12.32
8	LFU_WebServer_100	0	93.7	99.8	0.2	11.42

Columns 3 to 6 in Table 5 above show the percentage of proxy-processing latencies that fall within the specified time ranges (in milliseconds). For example, for Run 1 (LRU scheme with remote proxy and cache size of 50), 14.5% of all requests were satisfied with an observed latency less than 1 millisecond (at the proxy); for Run 8 (LFU scheme with proxy on the Web server and cache size of 100), all the observed latencies were greater than 1 millisecond (with 93.7% of them less than 10 ms). The last column in the table shows the average latency incurred at the proxy during each simulation run (for all 14000 requests present in the workload).

From the average latencies in the last column, the following observations are made: First, the LFU caching eviction scheme outperforms the LRU scheme (the latter showed higher average latencies). Second, for both caching schemes, the average latencies were lower when the proxy was located on the Web server machine versus its location as a 'remote' server. This behavior is expected because with the proxy server residing on the same machine as the Web server, the latency incurred at the proxy in fetching Web content and receiving the response is lower (due to the physical proximity of the proxy and server); as opposed to a remote proxy, which is located on a different machine between the client and Web server (and consequently further away from the Web server). Third, from the average latencies in the last column, the size of the cache did not influence the values as much as could be expected. It is generally expected that the bigger the cache size, the lower the latency at the proxy since more previously accessed documents are stored at the proxy for potential future requests. The average latencies in the last column do not clearly spell out this trend, except in Runs 7 and 8 where there is a slight decrease in the average latency when the cache size is 50 and 100, respectively.

5.1.5 Comparison to the Client Baseline architecture performance

Recall that in the Client baseline architecture (presented in Chapter 3), the mobile client Browser was responsible for the user interface management, in addition to the computationally intensive tasks of fetching requests from Web servers, parsing and rendering the content to the device. This proved to be very inefficient and impractical as described in Section 3.3.3.

Although the workload used in evaluating the baseline architecture is different from that used in this experiment, by observing the overall performance of both experiments we see that the MoBed architecture (with a caching proxy) is significantly better than the baseline architecture. Figure 16 represents the time taken to fetch and render the content of requested pages in the baseline architecture. It can be observed from this graph that the time taken to fetch and render all pages was no less than 10,000 milliseconds (ms). The bulk of this time was used up in fetching and parsing the page contents, and little time used in rendering the parsed HTML to the screen. Using the MoBed architecture, in this experiment, it was observed that for all eight simulation runs (using both caching policies), that over 99% of requests were satisfied in less than 500 ms. This means that the time taken to deliver the transcoded content bytes to the client was less than 500 ms. The 'fetch' stage is more time-consuming and memory intensive than the 'render' stage (recall Section 3.3.3). This is the case because the 'fetch' phase involves: reading data from the connection to the resource, creating HTML node objects and storing them in a vector for rendering. The 'render' phase simply entails iterating through all the HTML nodes created in the fetch phase, and appending their relevant text information on the mobile client screen object. The time taken to render a requested page (using the baseline architecture) was usually observed to be 5 to 15%(or less) of the total time taken to complete the request. As such, although the time measured in this experiment did not include the time taken to render the content on the client screen as it does in Figure 16, the time savings incurred when satisfying client requests are evident. This is because the presence of the MoBed cache reduces the time taken to fetch and transform a request to zero, when it is located in the cache.

5.2 Experiment 2: Data compression using the Proxy Transcoder

The goal of this experiment was to investigate the usefulness of the Proxy Transcoder as a data compression tool by determining the difference in the number of Web content bytes before and after the transcoding process.

5.2.1 Objective

Recall that the Proxy Transcoder was used for content adaptation, i.e. it was used to convert the parsed HTML content received from Web Servers to a simpler, compressed format for a mobile device. This experiment provided a comparison between the number of bytes received at the proxy from the original server, and the number of transcoded (simplified) bytes supplied to the mobile from the proxy server. In essence, this experiment provided a means of determining how much data compression was carried out by the Proxy Transcoder. Refer to Section 4.2.3 for details on the proxy transcoding process, as well as the generation of the converted web content using *'PageNodes'*.

5.2.2 Workload description

The same workload used in Experiment 1 was used in this experiment. It consisted of server traces collected from the CMPUT 301 course website at the University of Alberta in the Fall Semester of 2001. The traces were made up of 14, 000 requests generated from repeated accesses to 116 unique URLs from the course site.

The Proxy maintained a collection of all the unique URLs received through client requests. As the proxy satisfied client requests using this workload, 116 URLs were processed and statistics collected on the size of the requested content before and after the transcoding phase.

5.2.3 Experiment Design

In this experiment, statistics were recorded on two control variables: the size (in bytes) of the original data content received from the Web server by the proxy and the size (in bytes) of the transcoded data supplied to the mobile from the proxy server. These two statistics were documented for each unique URL that was requested by the client.

5.2.4 Results and Evaluation

The graph in Figure 31 shows the number of bytes received from the Web server versus the number of bytes transformed and sent to the mobile client from the proxy for all 116 unique URLs.

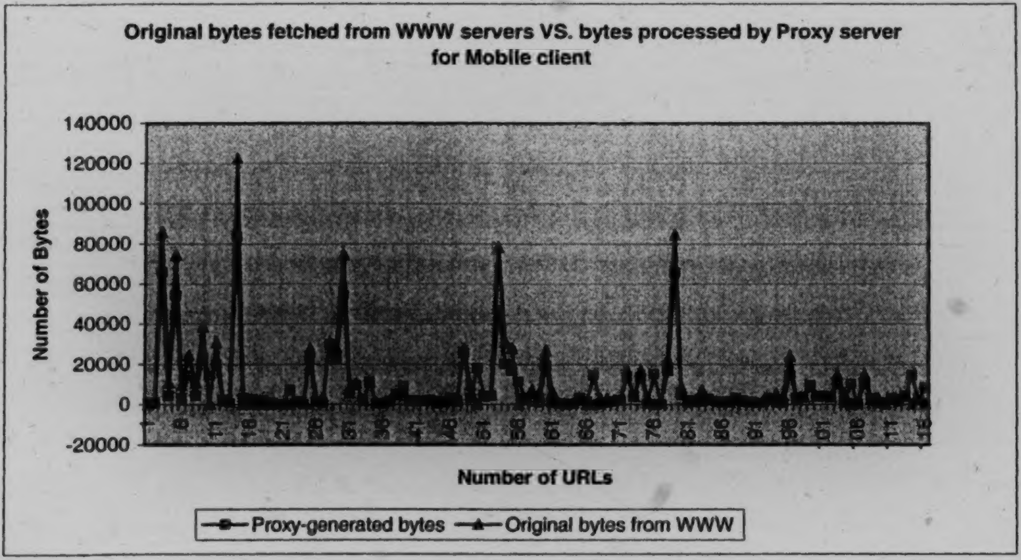


Figure 31 - Original bytes downloaded from Web servers Vs Proxy-transcoded bytes

From the graph, we can observe that in cases where the data content for the requested page was over 20kB, the effect of the proxy byte transformation process proved to be more noticeable and beneficial to the client (as less bytes are sent to the client, compared to the original downloaded size). For small byte sizes, the effect of converting the downloaded content sometimes results in a slightly larger byte content for the mobile client. This slight increase can be accounted for by the encapsulation overhead incurred in transforming bytes for the mobile client.

5.2.5 Comparison to the Client Baseline architecture performance

From Section 5.2.4, it is observed that although the proxy data compression scheme could be improved, there are some benefits to be reaped for performing content adaptation at the proxy (as opposed to the mobile client).

With the HTML parser used in both the Client baseline and MoBed architectures, when a page is parsed, HTML nodes are generated for HTML tags that are potentially useless to a mobile client, such as HTML remark tags and end tags. The HTML nodes generated from such tags do not provide any information that will be displayed to the user. Therefore, parsing and storing such nodes on a device with limited disk space is an expensive, and time-consuming task. When parsing is done on the Proxy, only HTML nodes that are useful and displayable to the client are transcoded and sent to the mobile. This process removes the unnecessary HTML from a requested web page before the user ever sees it. More benefits could be harvested with a proxy Transcoder that can compress the useful web data to much smaller sizes than are achieved by the current scheme.

5.3 Experiment 3: Caching at the Client-level while Prefetching at the Proxy

As previously mentioned, this testbed is used to investigate intelligent solutions for providing web content to mobile clients with reduced client-perceived latency. Experiment 1 addressed caching at the proxy level, using two cache replacement policies (LRU and LFU). The results demonstrated the need to explore another scenario for experimentation. The goal of this experiment was to assess the performance of MoBed with the following features: a cache maintained on the client; an infinite, local cache at the proxy, as well as prediction scheme implemented on the proxy. This assessment was achieved by means of a study consisting of trace-based simulation experiments run on the proxy server.

5.3.1 Objective

Before a detailed description of this experiment is provided, recall that: the proxy prediction engine (described in Section 4.2.4.1), builds a path tree to store path profiles of past user session traces and predicts a user's next request based on past users who behaved similarly. Also recall that the size of the path tree is controlled by a threshold value (henceforth referred to as *T-value*), which restricts the degree of expansion in the path tree. When the prediction engine is initiated, it is provided with a *training set* of past user sessions, from which it creates the path tree, and path profiles. A *test set* is then used to supply different user requests, for which the engine predicts the next move the user is likely to make.

The purpose of this experiment was to investigate the following:

1. *Experiment 3-1*: The impact of having a mobile client cache (with three different cache sizes: 8 kilobytes (kB), 30kB and 60kB) on the following response variables:

    - The percentage of '*No Proxy accesses*' for all clients (when requested item was found in the mobile cache),
    - The percentage of *prefetch-interrupts* resulting from attempting to prefetch documents which are larger than a client's available cache space. (A prefetch-interrupt was recorded when a client did not successfully receive the item predicted for its next request. Prefetching was interrupted when: there was an incoming request before it was completed (not enough time to complete the task), and when the file to be prefetched was larger than the user's specified available caching space.)
    - The total number of files prefetched to all clients, and
    - The total number of unique clients who received prefetched documents.

   The client cache size is the sole factor in this experiment, with the following levels: 8kB, 30kB and 60kB. The smallest cache size was taken to be 8kB because it is the minimum amount of non-volatile memory that can be allocated for application-created persistent data using the J2ME MID Profile. Newer J2ME-enabled phones on the market today display remarkable capabilities, such as increased processing speeds, heap sizes and shared memory for storage. Such devices can afford to have larger memory allocations for application-created data, hence the reason for investigating the impact of the two larger cache sizes of 30 and 60 kB.

2. *Experiment 3-2*: This study also investigated the accuracy of the predictions from the proxy's prediction engine. The prediction accuracy was measured using two response variables: the *predicted file hit ratio* and *predicted byte hit ratio*. There were three factors in this experiment that determined the response:
    - The path tree size (varied by changing the threshold value T),
    - Retraining the prediction engine using recently-accessed test requests, and
    - The size of the workload (using train/test datasets of varied sizes).

5.3.2 Workload description

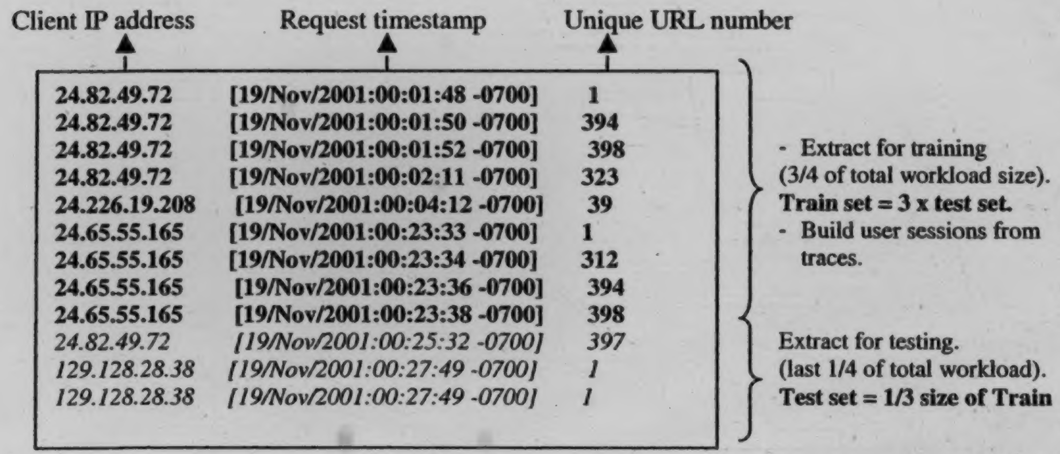
Two workloads were used for both Experiment 3-1 and 3-2. The first dataset was generated from server logs acquired from the CMPUT 301 course web site at the University of Alberta. These logs were collected over the Fall semester of 2001

(September to December) for a period of 122 days, resulting in just over 140000 requests on 405 distinct URLs. The requests generated from the first week of September and the last 2 weeks of December were not included in this dataset, as there was very little activity on the web site. This dataset is henceforth referred to as *C301*.

The second dataset was generated from server logs obtained from the Computing Science Department Web server at the University of Alberta. It consisted of the first 60000 requests extracted from server logs for December 5th 2003, with a total of 24,772 unique URLs. This dataset is henceforth referred to as *CS*. The two workloads differed from each other as follows. The C301 dataset was generated from logs from a small, course website with a small number of URLs and small client population. The CS dataset was generated from logs from a much larger departmental server that services a larger client population and where a larger number of URLs are accessed over a very short time period. These two workloads were chosen for this study because of their diversity from each other. When conducting a performance study for the purpose of investigating the effectiveness of caching and prefetching mechanisms, a smaller workload is expected to result in a better outcome than a larger one because: it is less likely to contain a lot of dynamic content, and contains fewer URLs (both of which are advantageous to caching). In addition, with a smaller workload, the chance of discovering access patterns from user requests is heightened (which could be beneficial to prefetching), as compared to a larger workload where requests are as good as random.

The accuracy of the proxy prediction engine was measured using training and testing data created from each workload. To simulate a practical application of these logs, a testing set was designed to contain only requests that occurred after all of the training set requests were collected. For example, if the requests used for training were gathered from September 1 to September 30, then the test requests used must be collected after that time (say, from October 1). If only one log was available from a given site, the log was used for the generation of both the training and testing set. The URLs in each log needed to be represented in a more compact format, to allow for easy storage and quick comparisons. As such, a unique integer was assigned to each unique URL present in the logs. Throughout this study, URLs are simply referred to using their unique integer identifiers.

As mentioned before, for each workload, smaller datasets of varied-sizes were created (henceforth referred to as *Workload partitions*). From these partitions, a few training and testing sets were produced, in order to investigate the effect of the train and test set sizes on the accuracy of the prediction engine. This heuristic was adopted to determine whether the prediction accuracy would improve as the number of processed requests increased. This heuristic simulates a real life setting, whereby the number of requests intercepted and processed by the proxy server accumulates over time (hence adding to the *access history* knowledge base). Workload partitions can be considered simply as varied-sized workloads obtained from the same source. For every training set, its corresponding testing set was taken to be a third of the size of the training set. For instance, consider a workload made up of 40 requests. Using this workload, the training set would consist of 30 requests, and 10 requests for its corresponding testing set. Figure 32 illustrates the process of generating training and testing sets from a sample workload partition.



```
24.82.49.72      [19/Nov/2001:00:01:48 -0700]    1
24.82.49.72      [19/Nov/2001:00:01:50 -0700]   394
24.82.49.72      [19/Nov/2001:00:01:52 -0700]    398
24.82.49.72      [19/Nov/2001:00:02:11 -0700]   323
24.226.19.208     [19/Nov/2001:00:04:12 -0700]   39
24.65.55.165     [19/Nov/2001:00:23:33 -0700]   1
24.65.55.165     [19/Nov/2001:00:23:34 -0700]   312
24.65.55.165      [19/Nov/2001:00:23:36 -0700]   394
24.65.55.165      [19/Nov/2001:00:23:38 -0700]   398
24.82.49.72       [19/Nov/2001:00:25:32 -0700]   397
129.128.28.38    [19/Nov/2001:00:27:49 -0700]   1
129.128.28.38    [19/Nov/2001:00:27:49 -0700]   1
```

- Extract for training (3/4 of total workload size).

Train set = 3 x test set.

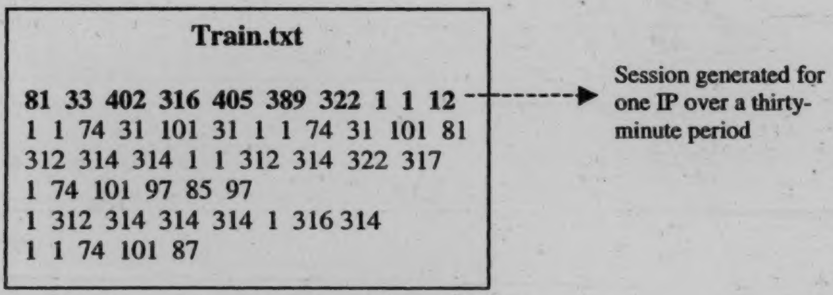
- Build user sessions from traces.

Extract for testing. (last 1/4 of total workload).

Test set = 1/3 size of Train

Figure 32 - Generating training and testing sets from a workload partition

Each training set consisted of a collection of complete user sessions generated from requests collected over a period of time. User sessions were made up of all requests issued by a given user within a thirty-minute period. This time heuristic was adopted for the following reasons. First, it is not possible for a single server to trace a user's paths through other sites; and thus impossible to verify if a user passed through another site between accessing two pages on a single server site [SKS98]. As such, a thirty-minute time period was adopted to ensure that any requests separated by more than thirty minutes constitute two separate user sessions. Figure 33 portrays a sample training set in a file called *Train.txt*.



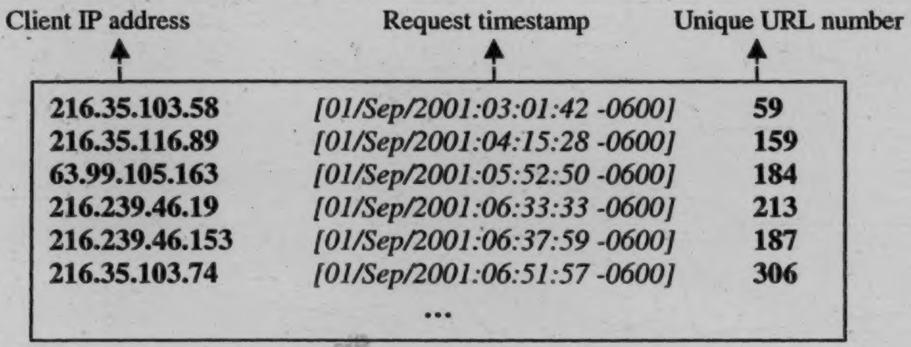
Train.txt

```
81 33 402 316 405 389 322 1 1 12
1 1 74 31 101 31 1 1 74 31 101 81
312 314 314 1 1 312 314 322 317
1 74 101 97 85 97
1 312 314 314 314 1 316 314
1 1 74 101 87
```

Session generated for one IP over a thirty-minute period

Figure 33 - A sample training set

Each testing set consisted of a list of client requests as they arrived at the server over time. Each request was made up of the IP address of the client requesting the data, the timestamp showing when the request arrived at the server, and the unique URL number for the request (as shown in Figure 34).



Client IP address	Request timestamp	Unique URL number
216.35.103.58	*[01/Sep/2001:03:01:42 -0600]*	**59**
216.35.116.89	*[01/Sep/2001:04:15:28 -0600]*	**159**
63.99.105.163	*[01/Sep/2001:05:52:50 -0600]*	**184**
216.239.46.19	*[01/Sep/2001:06:33:33 -0600]*	**213**
216.239.46.153	*[01/Sep/2001:06:37:59 -0600]*	**187**
216.35.103.74	*[01/Sep/2001:06:51:57 -0600]*	**306**
	...	

Figure 34 - A sample testing set.

Table 6 shows the different train/test sets generated from the *C301* logs; while Table 7 shows the train/test sets generated from the *CS* logs.

Table 6 - A summary of the train/test sets generated from the C301 workload partitions

Workload Partitions	Partition Size (# of Requests)	Number of TRAIN requests	Number of TEST Requests	Number of Unique URLs from Training set	MAX Number of Unique URLs from Testing set	Total number of Clients
1	1211	909	302	77	49	65
2	69,580	52,185	17,395	263	227	1129
3	109,523	82,356	27,167	307	241	1458
4	137,186	103,162	34,024	318	276	1416

Table 7 - A summary of the train/test sets generated from the CS workload partitions

Workload Partitions	Partition Size (# of Requests)	Number of TRAIN requests	Number of TEST Requests	Number of Unique URLs from Training set	MAX Number of Unique URLs from Testing set	Total number of Clients
1	6,276	4,677	1,599	3014	898	251
2	48,704	36,528	12,176	17,286	5944	1196

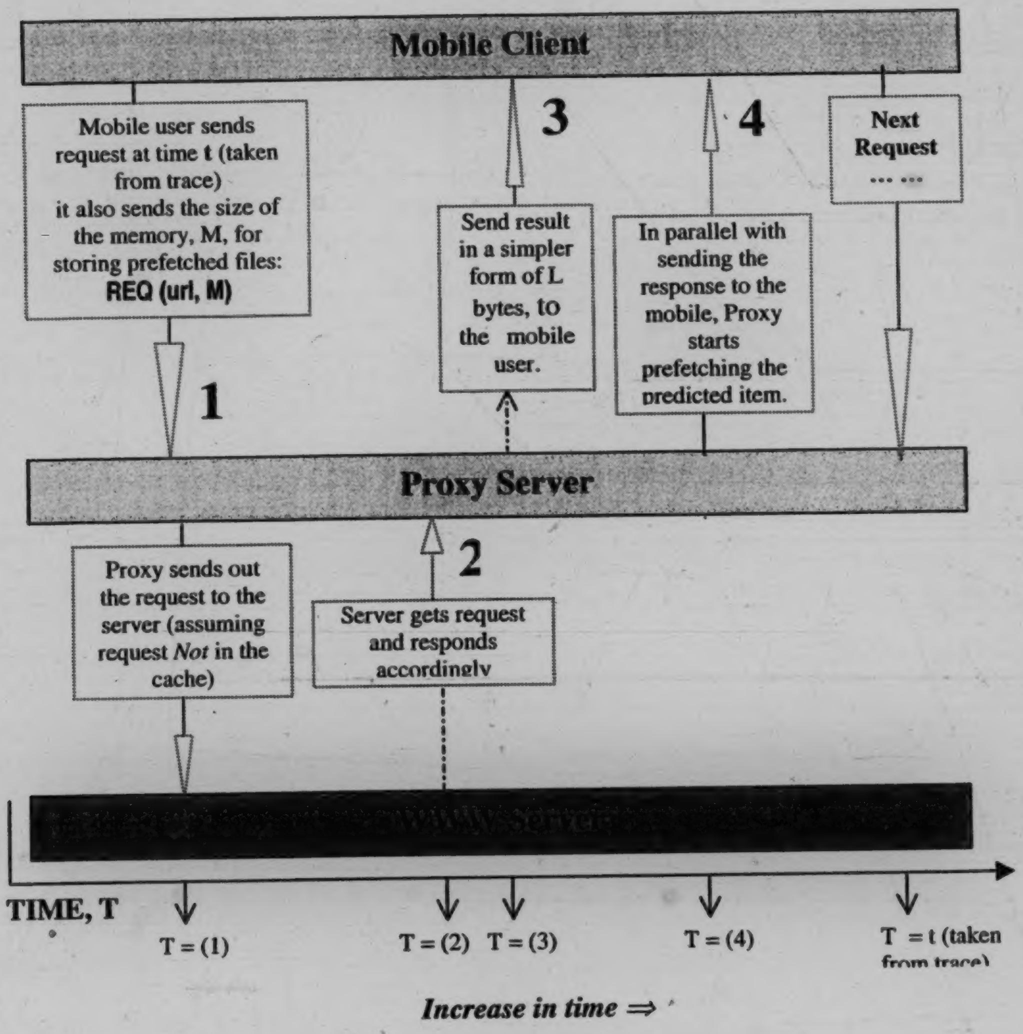
Tables 5 and 6 show the details on the train/test sets generated from the *C301* and *CS* logs, respectively. Column two represents the number of requests in each dataset generated from each workload; column three and four illustrate the size of the generated training and testing datasets; column five depicts the number of unique URLs obtained from the user sessions used in the initial training phase; column six shows the number of unique URLs obtained from the testing dataset; and column seven shows the total number of distinct clients serviced from the testing dataset only.

5.3.3 Structure of the Simulator

In this simulation study, the testbed is reduced to having one main component: the proxy server i.e. the client is simulated on the proxy and no separate client process is maintained. Since different workloads were used as a source of user requests, the proxy simply iterated through a test set generated from the workload, and issued requests on behalf of the client; thereby removing the need for explicitly defining a separate client process.

5.3.3.1 Time sequence illustration of a simulation run

An illustration of the time sequence simulation for each experiment run is provided in Figure 35. It captures the transfer of a requested file and any prefetched item to the client, before another client request is issued. Note that the proxy to mobile client link is bounded by a bit rate of 9600 bits per second. This bit rate was chosen because most minimum-footprint J2ME platform devices are characterized by connectivity to a wireless, intermittent connection with limited bandwidth of 9600 bits per second (bps) or less [RTV01].


Mobile Client
Mobile user sends request at time t (taken from trace) it also sends the size of the memory, M, for storing prefetched files: REQ (url, M)
1
3
Send result in a simpler form of L bytes, to the mobile user.
4
In parallel with sending the response to the mobile, Proxy starts prefetching the predicted item.
Next Request
... ...
Proxy Server
Proxy sends out the request to the server (assuming request Not in the cache)
2
Server gets request and responds accordingly
WWW Server
TIME, T
T = (1)
T = (2)
T = (3)
T = (4)
T = t (taken from trace)
Increase in time ⇒

Legend:

A. T.: Arrival Time of an entity to its destination;
request = The size of the request data from the mobile;
L = Number of transformed requested bytes;
9.6 kbps = The proxy-mobile link connection bandwidth

(1) A. T. = **t + request/9.6**
(2) A. T. = **t + request/9.6 + document_transfer_time** (doc_transfer_time)

(3) A. T. = **t + request/9.6 + doc_transfer_time + proxy_transform_time + L/9.6**
(4) A.T. = **t + request/9.6 + doc_transfer_time + proxy_transform_time + L/9.6 + prefetched_doc_transfer_time + prefetched_doc_transform_time + prefetched_doc_size/9.6**

Figure 35 - Time sequence illustration of a simulation run

Steps 1 to 4 shown in are described as follows:

- *Step 1:* The mobile user sends a request at time, *t* (extracted from the workload traces). It also advises the Proxy on the space it has available for storing prefetched files. The Proxy captures this request at *time (1),* shown in Figure 35. The Proxy then sends out the request to the Web server (if it can not satisfy it from its local cache).
- *Step 2:* The web server receives the request and responds with the requested content. The proxy receives the response at a *time (2)* (from Figure 35). The proxy then transcodes the original content to a version suitable for the mobile client.
- *Step 3:* The transcoded content results in a representation of *L bytes* that is dispatched to the client. The response arrives at the client at *time (3).*
- *Step 4:* In parallel with starting to send the response to the client (Step 3), the Proxy commences the prefetching process. If a document is chosen for prefetching, it is fetched, and transformed at the proxy. The time the prefetched item arrives at the client is determined by the times taken to fetch and transform the document content at the proxy. If it happens that the item is ready to be prefetched to the client before the initial client request has been satisfied, prefetching is suspended until the requested page transfer is completed. Under this assumption, the prefetched item finally arrives at the client at *time (4).*

5.3.4 Object and Data Structures of the Simulator

During each simulation run, there was no actual access to Web server resources when a client request was processed. For each unique URL in each workload, the following statistics were gathered on the Proxy during a pre-processing phase: the size of the mobile request (in the format: *<String: client_IP, int: URL id, Date: access_time>*), the time taken to fetch each URL, the transcoding time, the number of original response bytes from the Web server, and the number of proxy-transcoded bytes. A simple program was run on the proxy (iterating through each URL from a list of unique URLs in the workload) in order to collect the values for these attributes for each URL. These attributes were then encapsulated in an *URLUnit* object shown in Figure 36. As such, at the start of each simulation run, a collection of *URLUnits* corresponding to each unique

URL in the workload was available at the proxy. In calculating the simulation time when a request was being processed (illustrated in Figure 35), the URLUnit object pertaining to the requested URL was accessed and the relevant information retrieved.

The average time taken to read the data bytes from the Web server, create HTMLNodes and transcode the retrieved bytes was 14.13 milliseconds for the C301 workload, and 47.43 milliseconds for the CS workload. These averages hint at the complexity of the web pages retrieved for the URLs from the CS workload.

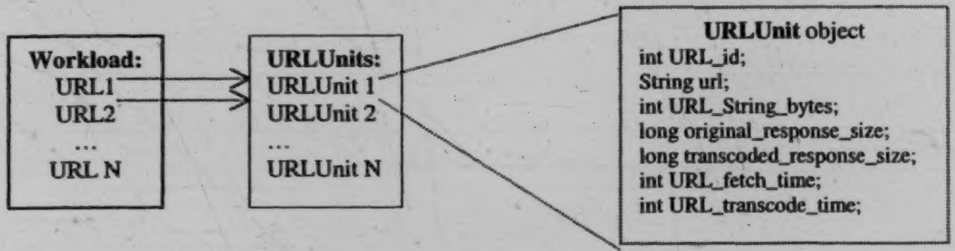

Workload:
URL1
URL2
...
URL N
URLUnits:
URLUnit 1
URLUnit 2
...
URLUnit N
URLUnit object
int URL_id;
String url;
int URL_String_bytes;
long original_response_size;
long transcoded_response_size;
int URL_fetch_time;
int URL_transcode_time;

Figure 36 - A URLUnit object

5.3.4.1 Client simulation on the MoBed Proxy server

The client is simulated on the proxy by means of the following proxy components:

1. *The Session Tracker Engine:*

The proxy's session tracker engine identifies different clients based on their IP addresses, while tracking each user's current session. A user session is taken to be all the requests issued by the user in a thirty minute time period. If a user makes only one request and never makes another, that request is considered to be the completed user session after thirty minutes elapse.

2. *The Mobile Cache Manager:*

This component manages a client cache (of a fixed size) on behalf of each unique client serviced by the proxy. When a client accesses the proxy for the first time, a cache is created for that IP. It is important to mention here that the client caches were used to store *only* items prefetched to the client from the proxy. Pages explicitly requested by the client are not cached. In a real-life setting, the decision to cache requested pages could be cached on the client by an intelligent Web browser, based on the user's interests. In this study however, a simple approach was taken by storing only prefetched items in the client caches. The *Mobile Cache Manager* is implemented as a Hash table, with unique client IP addresses mapped to their corresponding caches, as shown below in Figure 37. The class diagram in Figure 37 shows the interaction between the *MobileCacheManager*, *MobileCache* and *MobileCacheNode* classes on the MoBed Proxy.

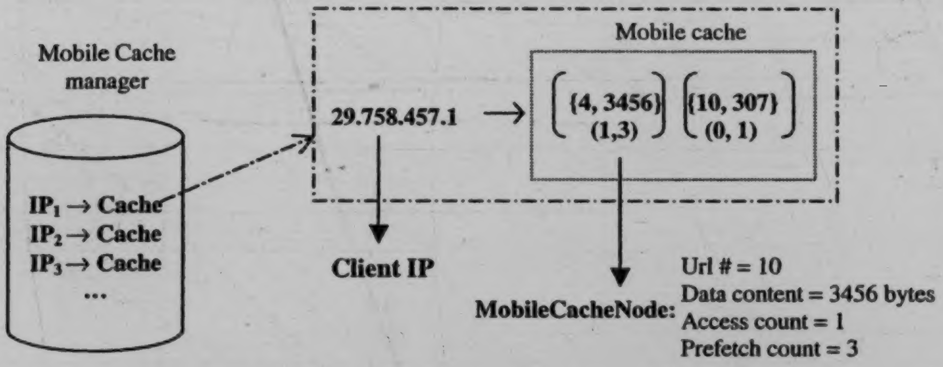

Mobile Cache manager
$IP_1 \rightarrow$ Cache
$IP_2 \rightarrow$ Cache
$IP_3 \rightarrow$ Cache
...
Mobile cache
29.758.457.1
{4, 3456} (1,3)
{10, 307} (0, 1)
Client IP
MobileCacheNode:
Url # = 10
Data content = 3456 bytes
Access count = 1
Prefetch count = 3

Figure 37 - An illustration of the function of the Mobile Cache Manager.

When a document is prefetched to a client, the data is stored in the cache in the form of a *MobileCacheNode* object (shown in Figure 37 above). The latter consists of the URL number (unique identifier for the URL string) and the number of transformed data bytes of the resource content. Each *MobileCacheNode* also has two important attributes: an access count and a prefetch count. The access count defines the number of times that node has been accessed from the cache; and the prefetch count defines the number of times that node was prefetched to the client. When a document is selected for prefetching, its corresponding *MobileCacheNode* is created with an access count of zero and a prefetch count of one. If a client cache is full, then the mobile cache object performs a simple, self-cleaning activity that removes all *MobileCacheNodes* that had been prefetched once and never accessed. As such, fresh pages are added when the size of the client cache reaches its maximum.

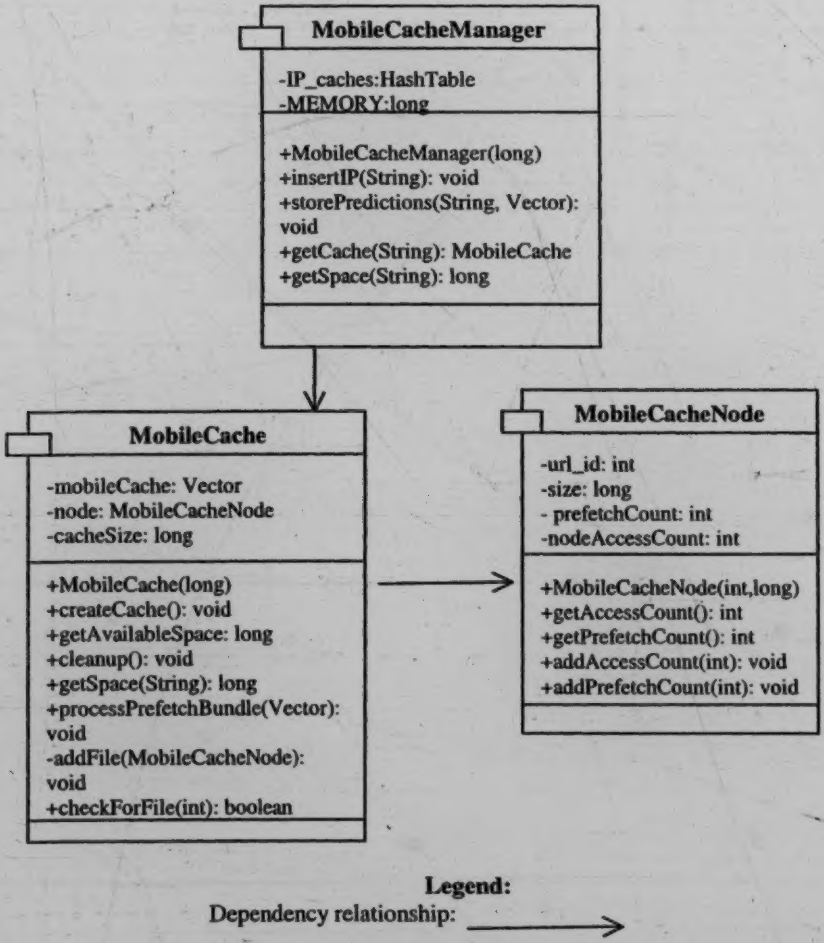

MobileCacheManager
-IP_caches:HashTable
-MEMORY:long
+MobileCacheManager(long)
+insertIP(String): void
+storePredictions(String, Vector): void
+getCache(String): MobileCache
+getSpace(String): long
MobileCache
-mobileCache: Vector
-node: MobileCacheNode
-cacheSize: long
+MobileCache(long)
+createCache(): void
+getAvailableSpace: long
+cleanup(): void
+getSpace(String): long
+processPrefetchBundle(Vector): void
-addFile(MobileCacheNode): void
+checkForFile(int): boolean
MobileCacheNode
-url_id: int
-size: long
- prefetchCount: int
-nodeAccessCount: int
+MobileCacheNode(int,long)
+getAccessCount(): int
+getPrefetchCount(): int
+addAccessCount(int): void
+addPrefetchCount(int): void
Legend:
Dependency relationship:

Figure 38 - Function of the Proxy MobileCacheManager component

5.3.4.2 The Simulator Control Flow

In all simulation runs in this experiment, a non-persistent infinite proxy cache was simulated i.e. the proxy cache contained every document ever accessed during any given run. At the start of each run, the proxy cache was empty and consequently populated as user requests were received and processed. There were two main stages in each run, namely the training phase and the testing phase.

1. *Training phase:*

At the start of this phase, the main storage units on the proxy were initialized, such as: the local cache, the client session tracker, and the client cache manager. The proxy controller class then read in user sessions from a training set, built a path tree and condensed it to a list of useful path list entries from which predictions would be later made (described in Section 4.2.4.1).

2. *Testing phase:*

With the completion of the training phase, the knowledge base for the prediction engine was ready and user requests could be processed. At the beginning of the testing phase begins, the test data was read from a file and saved into a collection of *Mobile Requests*, which consisted of three attributes: the IP address of the client, the time the request arrived at the server, and the unique URL number identifying the URL string requested. The proxy controller's *request-iterator method* iterated through the collection of *Mobile Requests*, and processed each request (as described below). Assume that the request being processed is from a client with IP address IP_1 at time T_1, requesting URL number 20.

The client's cache was examined to determine whether it contained a *MobileCacheNode* with URL number 20. Since the client was simulated on the proxy, the *Mobile Cache Manager* provided access to the cache belonging to IP_1, and supplied the value of the available client cache space (for storing prefetched items). If the document was found in the client cache, the proxy's *request-iterator* proceeded to release the next client request, and no prefetching was done for that request. This scenario was adopted because in a real-life situation, if a mobile client finds a requested document in its cache, it will have no need to access the proxy; consequently, no documents are prefetched to the client since the proxy never processed that request.

If the requested document is not found in the client cache, then the proxy's services are needed to satisfy the request. The proxy cache was verified to determine whether it contained that document. Note that since this was a simulation where the client was simulated on the proxy, the requested data was not actually sent back to the client. If the requested URL was found in the proxy cache or not, the time taken to satisfy that request was simulated (as shown in Step 3 in Figure 35); where the time taken to fetch the request from the Web was considered as 0 (if found in the cache) or the actual time taken to fetch the resource (if not found in the cache).

When a request was satisfied, another request was obtained from the *request-iterator* and processed and so on. It is important to mention here that prefetching only commenced if the proxy was idling after satisfying a request. That is, if there was a time interval between the time of the next client request and the completion of the current request, the proxy proceeded to predict the item for prefetching. If there was no time interval between consecutive requests, the next request was processed.

If the proxy had an idle time interval between consecutive requests, the proxy began the prediction process in parallel with starting to send the requested data response to the client. The prediction engine only predicted one document at a time (the best possible choice), given the user's current session and the available path profiles. Whether the prefetched item was found in the proxy cache or not, the simulated time taken to satisfy the prefetching process was calculated (as shown in Step 4 in Figure 35); where the time taken to fetch the predicted item from the Web was 0 (found in cache) or the actual time value. Note that the size of the prefetched item was restricted by the available cache space specified in the initial client request. If the prefetched document was too big, prefetching was interrupted. Prefetching was interrupted at any time, if there was an incoming client request.

In some of the simulation runs in this experiment, the prediction engine was *'retrained'* with requests from the test set after a fixed period of time elapsed (discussed in the upcoming section). The goal of *retraining* the prediction engine was to learn new paths, modify the path profiles, and perhaps improve the accuracy of the prediction engine. User sessions used for re-training were generated from the most recent test requests issued since the last retraining phase. If a simulation run contained a Retraining phase, retraining was performed at given time intervals during the testing stage. For example, the retraining could occur every 60 minutes during testing. This meant that following the timestamps of the URL requests issued, after processing requests for 60 minutes, the testing phase was suspended and retraining carried out. With the completion of the retraining phase, the test phase was resumed and other requests made and processed as already described above.

5.3.4.3 Experiment setup

Recall that the goal of this study was to assess the following (Section 5.3.1):

- *Experiment 3-1:* The effect of the client cache sizes (8, 30, 60 kilobytes) on the following dependent variables: the percentage of accesses NOT made to the Proxy when a request was made; the percentage of ***prefetch-interrupts*** from large prefetched files; the number of files prefetched to all clients; and the number of clients prefetched to.
- *Experiment 3-2*: The effect of the following factors on the prediction accuracy (predicted byte and file hit ratio) of the proxy prediction engine: the size of the Path tree used in making predictions (determined by the *T-value*), including a retraining phase for the prediction engine; and the workload size.

For these two experiments, several simulation runs were performed. Table 8 shows the experiment setup used for both experiments.

The following notation is used to refer to the factor levels in Table 8. The mnemonics used for the ***Retrain*** factor (Y and N) refer to the presence (Y) or absence (N) of a retraining phase in the prediction process. The ***Workload*** factor mnemonics used (C301 and GS) are suffixed by /1_2_3, /1_2 or /2 (e.g. C301/1_2_3) to denote the list of workload partitions used (generated from the named workload). For example, C301/1_2_3 means that partitions 1, 2, and 3 (shown in Table 6) from the C301 workload were used in that simulation run; CS/2 means that only the second partition from the CS workload (shown in Table 7) was used in the run.

Table 8 - Setup for Experiments 3-1 and 3-2

Number of Simulation RUNS	**T-Value**	**Client cache Size (kB)**	**Retrain**	**Workload**
24 runs	**T=2**	8	N	C301/1_2_3 CS/1_2
			Y	*C301/1_2_3*
		30	N	C301/1_2_3 CS/1_2
			Y	*C301/1_2_3*
		60	N	C301/1_2_3 CS/1_2
			Y	*C301/1_2_3*
27 runs	**T=3**	8	N	C301/1_2_3 CS/1_2
			Y	*C301/1_2_3* *CS/2*
		30	N	C301/1_2_3 CS/1_2
			Y	*C301/1_2_3* *CS/2*
		60	N	C301/1_2_3 CS/1_2
			Y	*C301/1_2_3* *CS/2*

As shown in Table 8, the experiment design for Experiments 3-1 and 3-2 is nested; i.e. in this experiment, the levels of one factor are chosen within the levels of another factor. For each T-value, the levels for the *Cache Size* factor are chosen (8, 30, 60 kB). Within each *Cache Size* level, the *Retraining* factor levels (Y and N) are chosen; and within each Retraining level, the different workloads are selected. For the T-value of 2, a total of 24 simulations were required; for the T-value of 3, a total of 27 simulations were required. Although the experiment design was same for both experiments, their response variables differed, as shown in Table 9.

Table 9 - Experiment 3: Factors and response variables

Experiment	Factors	Response variables
3-1	Client cache size	- % of Proxy accesses; - % of Prefetch-interrupts from large predicted files; - Number of files prefetched to clients; - Number of clients prefetched to.
3-2	- PathTree size (determined by T-value) - Using a Retraining phase - Workload size	- % of Predicted File Hit Ratios - % of Predicted Byte Hit Ratios

5.3.5 Results and Analysis

The results for Experiment 3-1 and 3-2 are presented and evaluated below:

5.3.5.1 Experiment 3-1

Recall that the goal of this experiment was to determine the effect of the client cache size on the following dependent variables: the percentage of accesses to the Proxy when a request is made; the percentage of *prefetch-interrupts* from large prefetched files; the number of files prefetched to all clients; and the number of clients prefetched to. Three levels were investigated for the Client cache size in this experiment (8, 30, 60 kB). It is important to mention that all clients were assigned the same maximum cache capacity throughout each simulation. For example, if the current cache size under investigation was 30 kilobytes, then all clients had a maximum storage capacity of 30 kilobytes. At the start of each experiment, all client caches were empty and populated in the course of the simulation. Table 10 and Table 11 summarize the results from the simulation runs using the response variables shown in Table 9 above, for the C301 and CS workloads respectively.

Table 10 - Experiment 3-1: Results obtained from the C301 workload

C301							
T-Value	**Client cache Size (kB)**	**Retrain**	**Workload Partition**	**% Of NO-Proxy Accesses**	**% Of Prefetch-interrupts from large files**	**Total # of files prefetched to all clients**	**Total # of Clients Serviced**
T=2	8	N	1	5.3	8.0	43	36
			2	23.1	29.1	1762	896
			3	25.2	34.8	2695	1160
		Y	1	*5.3*	*8.0*	*43*	*36*
			2	*23.3*	*30.0*	*1803*	*883*
			3	*25.6*	*35.4*	*2724*	*1156*
	30	N	1	6.0	1.4	46	38
			2	24.1	12.7	2147	920
			3	27.9	11.4	3434	1222
		Y	1	*6.0*	*1.4*	*46*	*38*
			2	*24.5*	*15.8*	*2149*	*915*
			3	*28.4*	*12.0*	*3483*	*1223*
	60	N	1	6.0	1.4	46	38
			2	25.0	3.1	2293	942
			3	28.3	3.6	3652	1253
		Y	1	*6.0*	*1.4*	*46*	*38*
			2	*25.4*	*3.4*	*2379*	*951*
			3	*28.8*	*4.0*	*3708*	*1254*
T=3	8	N	1	5.3	7.2	42	35
			2	23.7	25.9	1771	883
			3	25.4	32.0	2624	1151
		Y	1	*5.3*	*7.2*	*42*	*35*
			2	*23.8*	*28.2*	*1786*	*865*
			3	*25.9*	*32.6*	*2628*	*1151*
	30	N	1	5.6	1.5	45	37
			2	25.0	6.1	2155	911
			3	27.4	8.5	3336	1213
		Y	1	*5.6*	*1.5*	*45*	*37*
			2	*25.4*	*9.3*	*2158*	*892*
			3	*28.1*	*9.0*	*3357*	*1213*
	60	N	1	5.6	1.5	45	37
			2	25.0	1.6	2236	932
			3	27.7	3.5	3464	1244
		Y	1	*5.6*	*1.5*	*45*	*37*
			2	*25.7*	*2.2*	*2276*	*938*
			3	*28.4*	*3.8*	*3487*	*1244*

Table 11 - Experiment 3-1: Results obtained from the CS workload

CS							
T-Value	**Client cache Size (kB)**	**Retrain**	**Workload Partition**	**% Of NO-Proxy Accesses**	**% Of Prefetch-interrupts from large files**	**Total # of files prefetched to all clients**	**Total # of Clients Serviced**
T=2	8	N	1	23.2	0.0	1	1
			2	0.04	0.6	24	23
	30	N	1	23.2	0.0	1	1
			2	0.04	0.1	24	23
	60	N	1	23.2	0.0	1	1
			2	0.04	0.0	24	23
T=3	8	N	1	23.2	0.0	1	1
			2	0.07	0.7	22	22
		Y	2	*0.07*	*0.8*	22	22
	30	N	1	23.2	0.0	1	1
			2	0.07	0.1	22	22
		Y	2	*0.0*	*0.07*	22	22
	60	N	1	23.2	0.0	1	1
			2	0.07	0.0	22	22
		Y	2	*0.0*	*0.0*	22	22

The results summarized in the above tables are evaluated in the following sub-sections based on each of the response variables for this experiment:

5.3.5.1.1 The percentage of requests satisfied from client caches (No Proxy accesses):

Recall that: if a client's request was found in its local cache, it was instantly satisfied and there was no need to connect to the proxy server. There is a relation between the size of client caches and the number of accesses made to the proxy for a request: the bigger the client cache, the more documents can be stored, the less frequently do potentially-useful documents need to be evicted from the cache (to store newer items) and hence, the higher the chance of finding a requested document in the cache.

Table 10 summarizes the results from the *C301* logs. Column 5 in this table shows the percentage of *No-Proxy accesses* i.e. the percentage of times when the client request was found in its local cache over the total number of requests issued from the testing set. For instance, for the C301 Workload partition of 3, with a *T-value* of 2, the percentage of *No-Proxy accesses* increased from 25.2 to 27.9 to 28.3% as the cache sizes increased from 8kB to 30kB and 60kB, respectively. The same trend is noticeable over all the workload partitions and *T-values*. It is also observed from this table that for each cache size (over all *T-values*), the percentage of *No-Proxy accesses* increased, as the workload partition sizes increased. This is the case because the larger the size of the workload partition, the higher the number of clients, and hence the increase in the number of requests found in client caches (hence *No-Proxy accesses).*

Table 11 summarizes the results from the *CS* logs. The percentage of *No Proxy accesses* remains the same over all *T-values* for the CS workload partition 1. The percentage of *No-Proxy accesses* is surprisingly high for this workload partition, given that there was only one item ever prefetched to one client. The reason for this is because the one item prefetched to the single client was requested repeatedly (and found in the client cache), and hence recorded as a *No-Proxy access hit*. For the second partition, however, the percentage of *No-Proxy accesses* is quite low for both *T-values*. This is the case because from the few files successfully prefetched to the clients, only a very small number was accessed.

5.3.5.1.2 The percentage of 'prefetch-interrupts' resulting large prefetch documents:

Recall that a prefetch-interrupt was recorded when a client did not successfully receive the item predicted for its next request. Prefetching was interrupted when: there was an incoming request before it was completed (with not enough time to complete the task), and when the file to be prefetched was bigger than the user's specified available caching space. The latter reason is of importance in this experiment, because it relates to the size of the client cache. Column 6 in both Table 10 and Table 11 shows the percentage of prefetch-interrupts resulting from the proxy attempting to prefetch a document that was bigger than the client cache constraints. In Table 10 (C301 workload), it can be observed that as the cache size increases, the percentage of prefetch-interrupts drops (for all Partitions and all *T-values*). For example, for C301 partition 2 and *T-value* 2, it can be seen that the percentage drops from 29.1 to 12.7 to 3.1% as the cache size increases from 8kB to 30kB and 60kB, respectively. It is important to mention here that, the fewer the number of prefetch-interrupts, the higher the chances of a client finding its desired request in the cache, and the higher the number of *No-Proxy accesses*.

In Table 11 (CS workload) however, the results appear peculiar, perhaps even puzzling at a glance. Before explaining the reason for this, recall that the main difference between the two workloads (*C301* and *CS*) resulted from the fact that they were collected from a course website and a departmental server, respectively. The former contained less traffic and fewer requests over time, as compared to the latter, which received thousands of requests in a matter of minutes. This difference comes through in these simulation results because for a very busy server, there is a marked reduction in the number of successful prefetchs to clients. This is the case because prefetching only occurred when the proxy server was idling between client requests. In Table 11, the percentage of prefetch-interrupts (from large prefetch documents) is 0% for CS Partition 1, over all cache sizes because there are simply not enough successful prefetch attempts (since the requests came from a busy server). For CS Partition 2, on the other hand, the percentage of prefetch-interrupts is seen to drop from 0.7 to 0% (for T-value 3) and from 0.6 to 0.1 to 0% (for T-value 2); as the client cache size is increased from 8 to 30 to 60kB. Due to the fact that the size of CS Partition 2 was considerably larger as compared to those in CS Partition 1, there were enough prefetch attempts to demonstrate the observed trend.

5.3.5.1.3 The total number of files prefetched to all clients.

It follows from the above explanations that the larger the client cache, the higher the number of files prefetched to clients. This trend is demonstrated in Column 7 of Table 10 (C301 workload). The number of files prefetched to all clients increases (over all *T-values*) as the size of the client cache is increased from 8kB to 60kB. From Table 11 (CS workload), only one item is prefetched for CS Partition 1 and over 20 items for CS Partition 2. The reason for this is as mentioned above: due to the density of the requests in the *CS* log, which were obtained from a very busy server.

5.3.5.1.4 The total number of unique clients who received prefetched documents.

Table 10 and Table 11 demonstrate a general trend: the larger the client cache size, the more clients are likely to receive prefetched documents. Also, the bigger the workload partitions, the larger the number of clients serviced, and hence the higher the number of clients to potentially receive prefetched files.

5.3.5.2 Experiment 3-2

For each simulation run, the accuracy of the prediction engine was calculated. Statistics were collected showing: the number of times predictions were made and how often the predicted item turned out to be the user's next request. It is important to mention that in this experiment, the prediction-accuracy rates (accuracy of predictions) were measured from the perspective of the client (client-based predictability), since client-perceived latency is one of the main concerns of MoBed. Keeping this in mind, the client-based prediction-accuracy rates excluded cases where a prediction was made but the user never requested another page (such as at the end of a user's session). In this situation, the user never requests another page and hence does not suffer any losses even if the prediction was wrong. All predictions were made within a client's user session i.e. after their first request and as long as they issued requests. As previously mentioned, predictions were made only when there was enough history to back-up the guess i.e. when the user's current session has been previously 'learned' by the prediction engine; and only one item was prefetched to any given client at a time. Even though this amounts to the proxy not prefetching too often, it ensures that the best possible guess for each client's next request is made (choosing the quality of predictions over the quantity). The accuracy of the predictions from the proxy's prediction engine was measured based on three factors: re-training the prediction engine using recently-accessed test requests, varying the path tree size by changing the threshold *T-value*, and workload size (using train/test datasets of varied sizes).

5.3.5.2.1 The impact of the Re-training phase

The reason for re-training the prediction engine was to update the path profiles used in generating guesses for users' next requests. The impact of retraining depended on two factors: the number of times re-training occurred and the size of the testing data used to generate the new requests for retraining. If the size of the retraining data was relatively large, then there was a greater possibility of updating the frequency counts of the history paths, and hence the path profiles. Recall that predictions were made using history paths with the highest frequency counts. As such, if the re-training phase did not result in an increase in the frequency counts for some paths, no new knowledge was 'learned' and the prediction-accuracy rates would barely change from those acquired from the same simulation run with no re-training phase. All the simulation runs were performed first without a re-training phase, and then repeated with re-training occurring after a specific period of time.

Table 12 and Table 13 below provide a concise summary of the prediction-accuracy rates obtained for all simulation runs using the workload partitions generated from the *C301* and *CS* workloads. For the *C301* partitions, retraining was performed every six hours i.e. test requests were gathered in six-hour intervals and used as the new data for re-training. For the *CS-Dec5* datasets, re-training was performed every thirty minutes, since these logs were collected from a busy server that received thousands of requests within minutes.

Table 12 - Experiment 3-2: Results obtained from the C301 workload

C301							
T-Value	**Client cache Size (kB)**	**Retrain**	**# of Re-trains**	**Size of Re-training set**	**Workload Partitions**	**% of Predicted File Hits**	**% of Predicted Byte Hits**
T=2	8	N	-	-	1	37.1	29.6
					2	24.0	16.0
					3	20.8	14.3
		Y	*1*	*136*	1	*37.1*	*30*
			69	*17014*	2	*24.1*	*16.5*
			103	*26689*	3	*21.3*	*14.8*
	30	N	-	-	1	37.1	29.6
					2	24.4	16.1
					3	21.0	14.0
		Y	*1*	*136*	1	*37.1*	*30*
			69	*17014*	2	*24.6*	*16.7*
			103	*26689*	3	*21.5*	*14.4*
	60	N	-	-	1	37.1	29.6
					2	24.3	16.1
					3	21.0	13.6
		Y	*1*	*136*	1	*37.1*	*29.6*
			69	*17014*	2	*24.5*	*16.2*
			103	*26689*	3	*21.6*	*14.0*
T=3	8	N	-	-	1	40.6	32.3
					2	24.1	24.1
					3	21.6	21.6
		Y	*1*	*136*	1	*40.7*	*32.2*
			69	*17014*	2	*26.4*	*18.9*
			103	*26689*	3	*22.1*	*15.4*
	30	N	-	-	1	40.6	32.3
					2	24.4	16.3
					3	21.6	14.5
		Y	*1*	*136*	1	*40.6*	*32.3*
			69	*17014*	2	*26.6*	*18.9*
			103	*26689*	3	*22.2*	*15.0*
	60	N	-	-	1	40.6	32.3
					2	24.4	16.4
					3	21.6	14.1
		Y	*1*	*136*	1	*40.6*	*32.3*
			69	*17014*	2	*26.5*	*18.7*
			103	*26689*	3	*22.2*	*14.7*

Table 13 - Experiment 3-2: Results obtained from the CS workload

						CS	
T-Value	Client cache Size (kB)	Retrain	# of Re-trains	Size of Re-training set	Workload Partitions	% of Predicted File Hits	% of Predicted Byte Hits
T=2	8	N	-	-	1	36.8	35.8
					2	67.9	73.6
	30	N	-	-	1	36.8	35.8
					2	67.9	73.6
	60	N	-	-	1	36.8	35.8
					2	67.9	73.6
T=3	8	N	-	-	1	35.7	22.3
					2	71.3	78.1
		Y	*3*	*7065*	*2*	*71.5*	*78.1*
	30	N	-	-	1	35.7	22.3
					2	71.3	78.1
		Y	*3*	*7065*	*2*	*71.5*	*78.1*
	60	N	-	-	1	35.7	22.3
					2	71.3	78.1
		Y	*3*	*7065*	*2*	*71.5*	*78.1*

From Table 12 and Table 13, Columns 5 and 6 show the observed percentage of predicted file hits and byte hits (respectively). A *predicted file hit* is recorded when the predicted item for a user is actually requested next. The percentage of predicted file hits is defined as:

(The total number of correct guesses ÷ the total number of guesses) times 100.

Similarly, a *predicted byte hit* is recorded whenever there is a prefetch file hit. The percentage of prefetch byte hits is defined as:

(The number of requested predicted bytes ÷ the number of predicted bytes) times 100.

Note that every predicted file was not necessarily prefetched to the client, as prefetching was only successful if there was sufficient time to deliver the predicted item to the client before another request was received. Columns 7 and 8 show the number of times when retraining occurred and the total number of retraining requests used in the process, respectively.

Table 12 contains the prediction-accuracy rates observed for the *C301* workload. From the table, the following observations can be made. When re-training is performed, there is a general increase in the percentage of predicted file and byte hits over all three C301workload partitions, even though the difference may be small. For instance, for *T-value* 2 and a client cache of 8 kilobytes, for the C301 Partition 1, the percentage of predicted file hits remains constant at 37.1% with and without retraining. This is because of the small size of the dataset, and the small number of *retrains* performed (only one), with only 136 retrain requests. For this same partition, the percentage of predicted byte hits increases from 29.6 to 30% when retraining is executed. For C301 Partition 1, the percentage of predicted file hits increases from 20.8 to 21.3% and the percentage of predicted byte hits increases from 16 to 16.5% when retraining is performed 69 times using over 17000 retrain requests. In general, there is a slight increase in the predicted file and byte hits when retraining is executed, for all three partitions.

Table 13 contains the prediction-accuracy rates observed using the *CS* workload. Note that there was no retraining executed for CS partition 1, because it consisted of only 1599 test requests (as shown in Table 7), which were collected in a time period of less than ten minutes. Recall that for the *CS-Dec5* datasets, re-training was performed at thirty-minute intervals. From the table, it can be observed that: For the CS partition 1, the percentage of predicted file and byte hits remains constant over all cache sizes for both *T-values* when there is no-retraining. For CS partition 2 however, there is a slight increase in the percentage of predicted file hits from 71.3 to 71.5% when retraining is performed 3 times with over 7000 requests.

5.3.5.2.2 The effect of varying the T-value

Recall that in building a path tree from past user sessions, the number of potential paths in the tree could be controlled by the *T-value*, which restricts the expansion of every node in the path tree. A node in the tree is only expanded when its *maximal-prefix* has occurred at least T times [SKS98]. The maximal prefix of a path is the ordered sequence of all the URLs in the path minus the last one. Thus, the *T-value* is a threshold value that can be configured based on the available memory resources [SKS98].

For the *C301* workload, using Table 12, a comparison can be made between the prediction-accuracy rates over all C301 partitions, based on the *T-value*. It is observed from the table that the highest predicted file and but hit percentages are reached when the

T-value is 3. This is true over all the partitions and client cache sizes. To better explain this trend, consider Table 14 below, which shows the prediction-accuracy rates observed from 2 different *T-values* using the three C301 partitions (with a cache size of 8kB, and No Retraining phase).

Table 14 - Prediction-accuracy rates observed from 2 different T-values using three C301 partitions (With a cache size of 8kB, and No Retraining phase).

T Value	Client cache Size (kB)	Retrain	# of Re-trains	Size of Re-training set	Workload Partitions	% of Predicted File Hits	% of Predicted Byte Hits
T=2	8	N	-	-	1	37.1	29.6
					2	24.0	16.0
					3	20.8	14.3
T=3	8	N	-	-		40.6	32.3
					“	24.1	24.1
						21.6	21.6

Table 14 shows that the predicted file hit percentage rises from 37.1 to 40.6% (for the first partition) when the T-value changes from 2 to 3 respectively; while the predicted byte hit percentage rises from 29.6 to 32.3%. This increase can be credited to the size of the path tree created for each *T-value*. The higher the *T-value*, the smaller the size of the tree, since the expansion of every node in the tree is restricted, resulting in a smaller number of branches from each node. The smaller the number of branches from a node, the higher the probability of each branch occurring. Therefore, when the T-value is 3, there is an overall higher possibility of having better prediction-accuracy rates. It is also essential to point out here that when predicting a user's next request, the prediction engine only selects one chosen path from the list of path profiles – the path list entry with the same path as the client's current session and with the highest frequency count; That path list entry's *predictor* element is selected for prefetching. If there exist many other list entries with the same path and same frequency count (but different predictors), only one is chosen (the first one) since there is an equal probability for each path to occur. The fewer the number of path list entries with the same path and same frequency count, the higher the prediction rate.

5.3.5.2.3 The workload size (with varied train/test dataset sizes)

Recall that both workloads used in this study were divided into *workload partitions* that were in turn divided into train/test sets of varied sizes (recall Table 6 and Table 7). The results showing the impact of the workload size on the prediction-accuracy rates are summarized in Table 15 and Table 16:

Table 15 - Experiment 3-2: Prediction-accuracy rates observed from all four C301 partitions (with and without retraining).

C301					
Partition	**# of TRAIN requests**	**# of TEST Requests**	**Retrain**	**% of Predicted File Hits**	**% of Predicted Byte Hits**
1	909	302	N	37.1 – 40.6	29.6 – 32.3
			Y	37.1 – 40.7	30.0 – 32.3
2	52,185	17,395	N	24.0 – 24.4	16.0 – 16.4
			Y	24.1 – 26.6	16.5 – 18.9
3	82,356	27,167	N	20.8 – 21.6	14.0 – 15.0
			Y	21.3 – 22.2	14.4 – 15.4
4	103,162	34,024	N	27.8	17.8 – 19.1
			Y	28.5	19.4 – 22.6

Table 16 - Experiment 3-2: Prediction-accuracy rates observed from both CS partitions (with and without retraining).

CS					
Partition	**# of TRAIN requests**	**# of TEST Requests**	**Retrain**	**% of Predicted File Hits**	**% of Predicted Byte Hits**
1	4677	1,599	N	36.8	35.8
			Y	-	-
2	36,528	12,176	N	71.3	78.1
			Y	71.5	78.1

In Table 15, columns 5 and 6 show the range of file and byte hit percentages (respectively) for each partition, over all T-values and client cache sizes, i.e. they portray the lowest and highest rates ever obtained using that workload partition (with and without retraining). For the *C301* partitions, the general trend observed is that the bigger the partition, the lower the prediction-accuracy rates (both file and byte hit percentages).

For the CS workload, the results in Table 16 show that both file and byte hit percentages are surprisingly high for the second larger CS partition than for the first, especially given that the complete CS workload contained over 17000 distinct URLs in its training data alone. The high prediction-accuracy rates observed for this dataset can be accounted for as follows. The large number of distinct URLs in the training data resulted in the creation of a broad path tree during the training phase. The breadth of a path tree increases as the number of children of the tree's root node increases. Note that the number of child nodes of the root node cannot exceed the total number of unique URLs present in the training data. The CS partition 2 contained 36,528 train requests, consisting of 17,286 unique URLs (shown in Table 7), showing that the path tree created from this workload would be broad with little depth and limited branching. The fewer the branches from each node in the tree, the higher the probability of guessing one of its child nodes, resulting in increased prediction rates.

5.3.6 Comparison to the Client Baseline architecture performance

The MoBed architecture in this experiment is characterized by: caching support on the mobile client; an infinite cache at the proxy; as well as prefetching functionality. It is not trivial to analyze the benefits incurred from the different experiments carried out using MoBed, due to the diversity in the nature and design of the workloads and experiments. However, from the results and evaluation of Experiments 3-1 and 3-2 presented in Section 5.3.4 above, it can be observed that caching and prefetching at the MoBed proxy level is a promising combination when studying the performance of an intercepting proxy for satisfying client requests.

This MoBed architecture undoubtedly outperforms the client baseline architecture. First, the MoBed architecture attempts to reduce client-perceived latency from two perspectives: maintaining a client cache, infinite proxy cache, in addition to prefetching probable future requests from the proxy. Any results obtainable from such attempts are definitely an improvement to those reachable by the naïve client baseline architecture. Although not explicitly calculated, the performance of the MoBed architecture implemented in Experiments 3-1 and 3-2 is an improvement to that used in Experiment 1, because there was no caching policy implemented in the former, allowing all requests ever made to be available at the Proxy for all other clients (thereby increasing the chances of finding a request in the Proxy cache).

In general, the MoBed architecture implemented in Experiments 3-1 and 3-2 demonstrated the influence of the following factors on the perceived latency after client requests: the content adaptation (transcoding) scheme utilized at the proxy, the presence of a client and proxy cache, as well as the use of a prefetching mechanism at the proxy.

5.4 Summary

The simulation study presented in this chapter was designed to investigate the functions of the MoBed client-proxy server architecture in providing Web access for J2ME-enabled devices.

- The goal of Experiment 1 was to investigate the benefit of introducing caching only at the proxy level, using two caching schemes: LRU and LFU. The general conclusion was that the LRU replacement scheme performs slightly worse than the LFU scheme, and the proxy latency was observed to be higher with a remote proxy, as opposed to the proxy located on the Web server machine (the difference in the machines used for both the remote proxy and Web server may have influenced this result).
- Experiment 2 investigated the data content adaptation or Transcoding process carried out at the proxy in order to compress data to be sent to the mobile.
- Experiment 3 was designed to assess the performance of MoBed with the following features: a cache maintained on the client; and a prediction and caching scheme implemented on the proxy. This assessment is achieved by means of a study consisting of trace-based simulation experiments run on the proxy server. This experiment investigated: the impact of having a cache on the mobile client, with three different cache sizes of 8 kilobytes (kB), 30kB and 60kB; the accuracy of the predictions from the proxy's *prediction engine* based on three factors: re-training the prediction engine using recently-accessed test requests, varying the path tree size by changing the threshold T-value, and the using train/test datasets of varied sizes.

The results observed from all the above experiments, although not very high in some cases, are reasonable and promising enough to justify further research on improving the

MoBed proxy's function in caching and/or prefetching. The main contribution of this research does not lie principally in the results of these experiments but in the testbed architecture design, which allowed for complete factorial, and nested experiment design.

Chapter 6 Conclusion and Future Work

6.1 Research Contributions

The main objective of MoBed is to provide an experimental test-bed for designing, developing and analyzing different caching and prefetching schemes that can be used in devising Web access solutions for J2ME-enabled devices. The main contributions of this thesis are outlined below:

- As mentioned above, the objective of MoBed is to investigate an intelligent method for flexibly combining caching and prefetching schemes towards providing Web access solutions for wireless devices. This project has achieved this objective while adaptively separating the mobile-resident from the proxy-resident functionality.
- J2ME is a fairly new specification that is rapidly growing in popularity. Such technology attracts research because it is still in its adolescence, providing ample room for growth and improvement. This research introduces a fresh perspective on mobile web access targeted specifically towards the J2ME platform.
- All the experiments conducted using this framework (discussed in Chapter 4) show that there are benefits to be gained for having a client - proxy architecture for wireless Web access: It was shown that a Browser cache on the mobile device can considerably reduce the observed delays when a page is requested, especially if it stores prefetched items received from the proxy. It was also shown that caching and prefetching at the proxy-level can be particularly advantageous to J2ME devices that possess more than just the minimum requirements for supporting the CLDC and MID profile.
- The main contribution of this research is not so much the results of the experiments, as the creation of a modular, configurable testbed architecture design for investigating mobile Web access solutions using caching and prefetching schemes.

In general, the results from the experiments conducted on this architecture are reasonable and justify further research on improving the MoBed proxy's function in caching and/or prefetching.

6.2 Future Work

Possible future work on MoBed include the following:

- MoBed is still in its beginning stages, and there is room for improving some of its components:
  - The HTML parsing scheme on the Proxy server can be changes to use a simpler, less time-consuming parser that produces parsed content with little overhead.
  - The packaging of converted requested bytes for the client sometimes results in the generation of a package larger in size than the original content downloaded from the Web server. This overhead is particularly noticeable when small HTML pages are requested. One possible area for future work could be to investigate other more efficient HTML compression schemes that can be used to 'simplify' the requested data before it is sent back to the client.
  - Other caching and prefetching schemes can be investigated and used for the proxy's prediction engine. At this stage of the research, only one prediction scheme was implemented and tested (prediction through path-profiling user history). Even though the prediction-accuracy rates observed in Chapter Four were fairly reasonable, there is potential for improvement by examining other prefetching algorithms.
- The simulation study performed on the MoBed proxy used two different workloads to investigate the functionality of the framework. This study could be carried out with workloads obtained from a number of sites that vary in size, content, etc. to ensure that the results obtained from experiments are not biased to the workload used.
- For the experiments carried out in this research, the proxy-mobile link is bound by a bit rate of 9600 kilobits per second at all times. At the moment, there is no particularly accurate model for simulating the transfer delays on the mobile to proxy link. This connection can be characterized by higher and lower speeds over time. At times, when the speed is reduced, the bottleneck may not be the wireless link, and hence the approximation of such delays needs to be simulated to include a random component from the wired, which could be a bottleneck at times. With an accurate transfer model in place, experiments could be designed to investigate the actual client-perceived latency using this framework – while keeping in mind the three main

sources of latency in this architecture: the proxy to client connection, the server to proxy connection, and the data conversion process at the proxy.

- At this stage of the research, the development of the client-browser has been restricted to the J2ME Wireless toolkit environment. This device emulator provides only an approximation of the physical device. Hence, results acquired from it regarding the execution or performance of a MIDlet, are not guaranteed to be identical to those obtained when testing on the actual device. The Browser application in this project has not been deployed to an actual J2ME-enabled device and tested in that setting. This is an important step for future work because MoBed can actually be tested for performance in a real-life setting. This may bring up other issues or concerns with the architecture that are not considered at this stage and would otherwise be unforeseen.
- In future, the MoBed proxy server could potentially service wireless J2ME clients that have different device capabilities and constraints. The proxy could then provide better service by 'discriminating' between clients i.e. by maintaining a knowledge base of the device capabilities of all its clients. One possible benefit of this could be reaped when prefetching for clients. The proxy could be fairly liberal when prefetching documents to a 'state-of-the-art' client with fairly large disk area, memory and processing speed; as compared to a client that has the minimum support for J2ME.
- Transforming MoBed to a testbed framework architecture provides a very promising direction for future work. Such a framework would be very useful in providing a testbed for the investigation of various caching and prefetching algorithms to support Web access for mobile clients. In designing such a framework, determining the correct extension points or 'hooks' is crucial, where a hook is a point in a framework that is meant to be adapted. Extension points for a MoBed Framework could be implemented for the following features: a caching and/or prediction scheme, a transcoding process (for data compression at the proxy), and HTML parsing functionality. These features are extension points because they define the usefulness of the testbed. Having the ability to adapt the framework through these hooks will mean that these features can be enabled, disabled, replaced, or modified, hence

fulfilling the goal of having a testbed; even though the transformation from MoBed to MoBed Framework is likely to be non-trivial.

6.3 Conclusion

Wireless devices like cell phones, two-way pagers, PDAs, etc. are popular in this day and age because they provide instant gratification and convenient services to users without restricting them to a particular place and time. Nowadays, such small devices support additional features like email access, messaging, address book and calendar services, as well as Web browsing. There is a rising need and importance for wireless Web access from portable devices and cell phones.

A lot of research has been reported on the performance of Web Caching and Prefetching for wired Internet access, but in a wireless network, Internet access is substantially different. The main objective of this research was to design a general test-bed for investigating different caching and prefetching schemes that can be used with mobile devices and the Java™ 2 Micro Edition platform (J2ME™); opening up a fresh perspective for providing Web access solutions for small, wireless devices.

This research has achieved its objectives through the implementation and experimentation of MoBed, which shows a lot of promise in the long-term as discussed in Section 6.2. It is hoped that this thesis will inspire further research in the study of possible Web solutions for wireless devices in general.

Bibliography

[ABQ01] Atul Adya, Paramvir Bahl, and Lili Qiu; Analyzing browse patterns of Mobile Clients; *Microsoft Research, Redmond, Washington, 2001*

[Akc01] Muammer Akcay; Advance Web requests: Reducing latency while utilizing resources more effectively via the Instructor initiated Prefetching; Department *of Electrical Engineering and Computer Science – Lehigh University; August 2001.*

[ASA+95] M. Abrams, C.R. Stanbridge, G. Abdulla, S. Williams, and E.A. Fox; Caching Proxies: Limitations and Potentials; *Proceedings of 4th International World Wide Web Conference, pages 119-133, Boston, USA, 1995.*

[Bat01] K. Bates; The Wireless Web is Approaching Adolescence; *January 2001*

[BGMP00] O. Buyukkokten, H. Garcia-Molina, and A. Paepcke; Seeing the Whole in parts: Text Summarization for Web Browsing on Handheld devices; *Stanford University, 2000*

[CCC+03] Claudia Canali, Valeria Cardellini, Michele Colajanni, Riccardo Lancellotti, and Philip S. Yu; Cooperative TransCaching: A System of Distributed Proxy Servers for Web Content Adaptation; *Poster Proc. of the Twelfth International World Wide Web Conference (WWW2003), Budapest, Hungary, May 2003*

[CFK95] Pei Cao, Edward W. Felten , and Anna R. Karlin , Kai Li; A Study of Integrated Prefetching and Caching Strategies; *1995*

[Cha95] Yatin Chawathe, A Load Balancing Resource Locator for Proxies; *University of California, Berkeley, 1995*

[Cha03] S. Chadha; J2ME Issues in the Real Wireless World; *MicroDevNet – Micro Java Network; 2003*

[CJ97] C.R. Cunha, and C.F.B. Jaccoud; Determining WWW User's Next Access and its Application to Pre-fetching; *In Proceedings of Second IEEE Symposium on Computers and Communications (ISCC'97), Alexandria, Egypt, July 1997.*

[CM03] H. Chen and P. Mohapatra; A Novel Navigation and Transmission Technique for mobile handheld devices; *University of California, Davis January 2003*

[CZ01] Xin Chen, and Xiaodong Zhang; Coordinated Data Prefetching by Utilizing Reference Information at Both Proxy and Web Servers; *Proceedings of the 2nd ACM Workshop on Performance and Architecture of Web Servers (PAWS-2001), 2001*

[CZ02] X. Chen, and X. Zhang; Popularity-Based PPM: An Effective Web Prefetching Technique for High Accuracy and Low Storage; *2002 International Conference on Parallel Processing (ICPP'02), 2002.*

[Dav02] Brian D. Davison; Predicting Web actions from HTML content; *Proceedings of the Thirteenth ACM Conference on Hypertext and Hypermedia (HT'02) 2002.*

[Dej99] Petkovic Dejan; Intelligent proxy caching based on the principles of temporal locality; *Department of Computer Engineering, School of Electrical Engineering, University of Belgrade, 1999.*

[Duc99] Dan Duchamp; Prefetching Hyperlinks; *AT&T Research Labs; Proc. 2nd Usenix Symp. Internet Technologies and Systems, Usenix, Berkeley, California, 1999.*

[EJM00] A.N. Eden, B.W. Joh, and T. Mudge; Web latency reduction via Client-side Prefetching; *Proceedings 2000 IEEE Int. Symposium on Performance - Analysis of Systems & Software (ISPASS-2000), Austin, TX, pp. 193-200; 2000.*

[FB96] Armando Fox and Eric A. Brewer; Reducing WWW Latency and Bandwidth Requirements by Real-Time Distillation; *University of California, Berkeley; Fifth International World Wide Web Conference Paris, France, May 6-10, 1996*

[HBA99] O. K. Hong, and F. Biuk-Aghai; A Web Prefetching Model Based on Content Analysis; *University of Macau; 1999*

[Hem02] David Hemphill; J2ME and J2EE: Together – "At Last Sun has developed a blueprint for creating mobile and wireless applications that access enterprise services—where do we go from here?", *April 2002 Issue.*

[JC98] Q. Jacobson, and P. Cao; Potential and Limits of Web Prefetching Between Low-Bandwidth Clients and Proxies; *Department of Electrical and Computer Engineering, Proceedings of the Third International WWW Caching Workshop, 1998*

[KKO98] Jussi Kangasharju, Young G. Kwon, and Antonio Ortega; Design and Implementation of a Soft Caching Proxy; *Computer Networks and ISDN Systems Integrated Media Systems Center, Los Angeles, 1998.*

[KLM02] Bjorn Knutsson, Honghui Lu, Jeffrey Mogul; Architecture and Pragmatics of Server-directed Transcoding; *Proceedings of the 7th International Workshop on Web Content Caching and Distribution, Boulder, CO, USA, August, 2002.*

[KLM97] T.M. Kroeger, D.D.E. Long, and J.C. Mogul; Exploring the Bounds of Web Latency Reduction from Caching and Prefetching; *USENIX Symposium on Internet Technologies and Systems, 1997*

[Mah01] Maheshwari, A.; TranSqui: Transcoding and Caching Proxies for Heterogenous E-Commerce Environments; *UM-CS-2001-051, December 2001*

[MC98] Evangelos P. Markatos, and Catherine E. Chronaki; A Top-10 Approach to Prefetching on the Web; *ICS Foundation for Research and Technology; 1998*

[McA02] S. McAteer; Java Will be the Dominant Handset platform; *MicroDevNet – Micro Java Network; 2002*

[Mor01] Micheal Morrison; Getting to know the J2ME Emulator; *Article courtesy of sampublishing*.com - *excerpted from Sams Teach Yourself Wireless Java w/J2ME in 21 Days; August 2001*

[NKM01] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos; Effective Prediction of Web-user Accesses: A Data Mining Approach; *Aristotle University; 2001*

[Nyl01] Kristian Nylund; Developing software for mobile phones using J2ME; *ACM Classification: D.2.0, D.2.3, ACM SIGs: SIGMOBILE, SIGSOFT; 2001*

[PM96] V.N. Padmanabhan, and J.C. Mogul; Using Predictive Prefetching to Improve World Wide Web Latency; *Proceedings of the ACM SIGCOMM '96 Conference 1996*

[PS01] M. Papadopouli, and H. Schulzrinne; Design and Implementation of a Peer-to-Peer Data Dissemination and Prefetching Tool for Mobile Users; *First NY Metro Area Networking Workshop, IBM TJ Watson Research Center, Hawthorne, New York, March 12th, 2001.*

[RP01] James Reilly, and David Price; Developing MIDP Client/Server Applications; *JavaOne SUN's 2001 Worldwide Java Developer Conference; 2001*

[RTV01] R. Riggs, A. Taivalsaari, and M. VandenBrink; Programming Wireless Devices with the Java 2 Platform, Micro Edition; *Addison Wesley 2001.*

[Sab97] K. Sabnani; Wireless Data Services; *Bell Laboratories NJ. Holmdel NJ 0773, March 1997*

[Sin03] Tony Sintes; Memory matters; *http://www.javaworld.com/javaworld/javaqa/2001-12/03-qa-1228-memory.html; January 2003*

[SKS98] S.E. Schechter, M. Krishan, M.D. Smith; Using Path Profiles to Predict HTTP Requests; *Harvard University Division of Engineering and Applied Sciences, and Microsoft Corporation Internet Server team; April 1998.*

[STHK03] Bill N. Schilit, Jonathan Trevor, David M. Hilbert, and Tzu Khiau Koh; m-links: An infrastructure for very small Internet devices; *Mobile Computing and Networking; December 2003*

[STRS02] Aameek Singh, Abhishek Trivedi, Krithi Ramamritham and Prashant Shenoy; PTC : Proxies that Transcode and Cache in Heterogeneous Web Client Environments; *In the Proceedings of The Third International Conference on Web Information Systems Engineering -(WISE), December 2002 (a Best Paper)*

[Sul01] A. Sullivan; J2ME: Why Now? *MicroDevNet – Micro Java Network; 2001*

[SUN03] Java Blueprints for a Wireless white paper - Designing Wireless Clients for Enterprise Applications with Java Technology*; June 2003*

Source Code and Internet Resources

[HP1.1] HTML Parser version 1.1; *http://htmlparser.sourceforge.net/* - New releases available at: http://htmlparser.sourceforge.net/javadoc_1_3/; *Last access to site: December 2003.*

[JIMI] JIMI Software Development Kit; Jimi – a class library for managing images. *http://java.sun.com/products/jimi/*; *Last access to site: July 2003.*

[JTT] J2ME Tech Tips: Wireless Tech Tips – Object Serialization in CLDC-based profiles : Persistence classes – VectorHelper, Persistent interface.*http://java.sun.com/developer/J2METechTips/2002/tt0226.html*; *Last access to site: June 2003*

[JW-J2ME] Java 2 Platform, Micro Edition (J2ME) Resources and Articles: *http://www.javaworld.com/channel_content/jw-j2me-index.shtml; Last access to site: January 2004.*

[JYL-JW] M. Juntao Yuan, and Ju Long; Cookie Support - Extended and enhanced for JavaWorld. *Source code: RMSCookieConnector class. http://www.javaworld.com/javaworld/jw-04-2002/jw-0426-wireless.html; Last access to site: December 2003.*

[LG-J2ME] The Lurker's Guide to J2ME; Why J2ME? *http://www.blueboard.com/j2me/why.htm*; *Last access to site: December 2003.*

[MD503] Timothy Macinta; MD5 implementation in Java; Fast implementation of RSA's MD5 hash generator in Java JDK Beta-2 or higher. *http://www.twmacinta.com/myjava/fast_md5.php; 2003; Last access to site: July 2003.*

[SUN] Java Sun website for Java Technology – *http://java.sum.com* and Java Technology, Java 2 Platform, Micro Edition (J2ME) – *http://java.sum.com/j2me/; Last access to site: January 2004.*

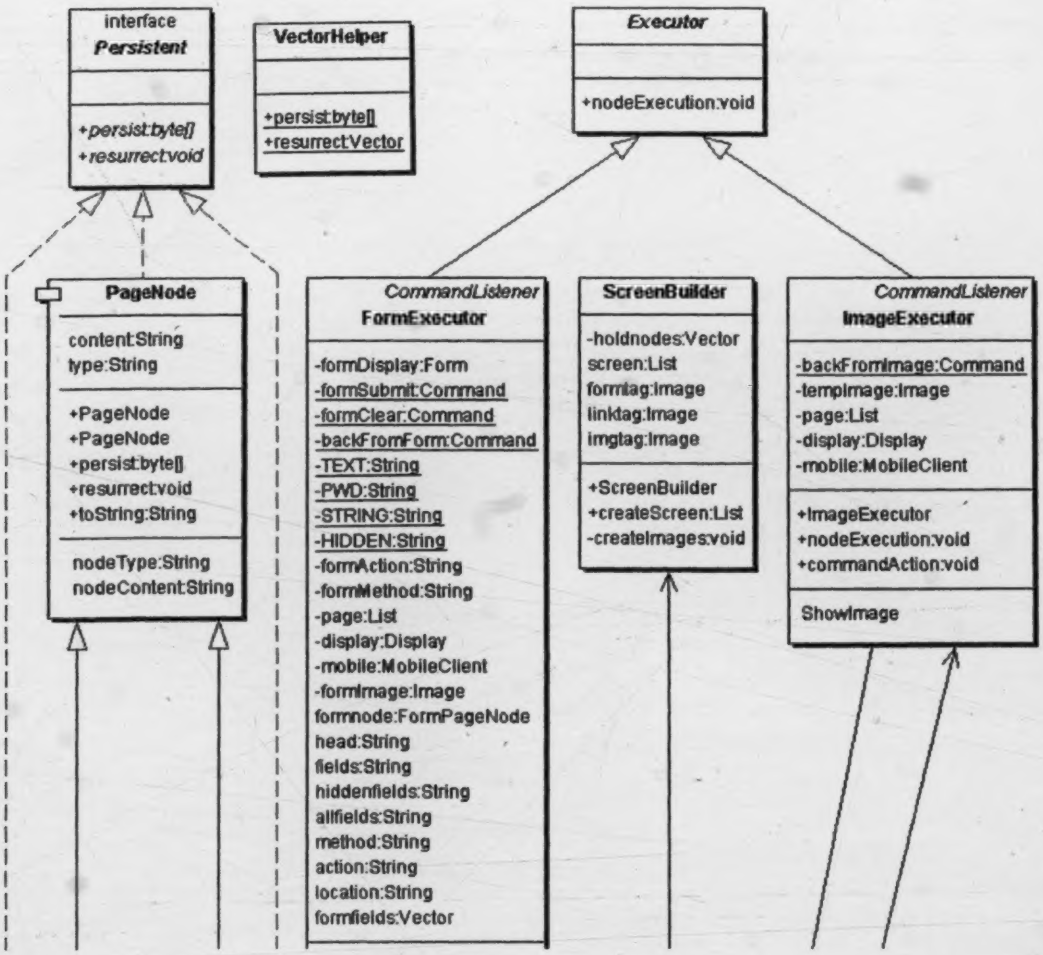
Appendices

The class names shown in the class diagrams may not be identical to the names of the components described in the thesis. As such, before each class diagram is provided, a small description is given to show the use of each class.

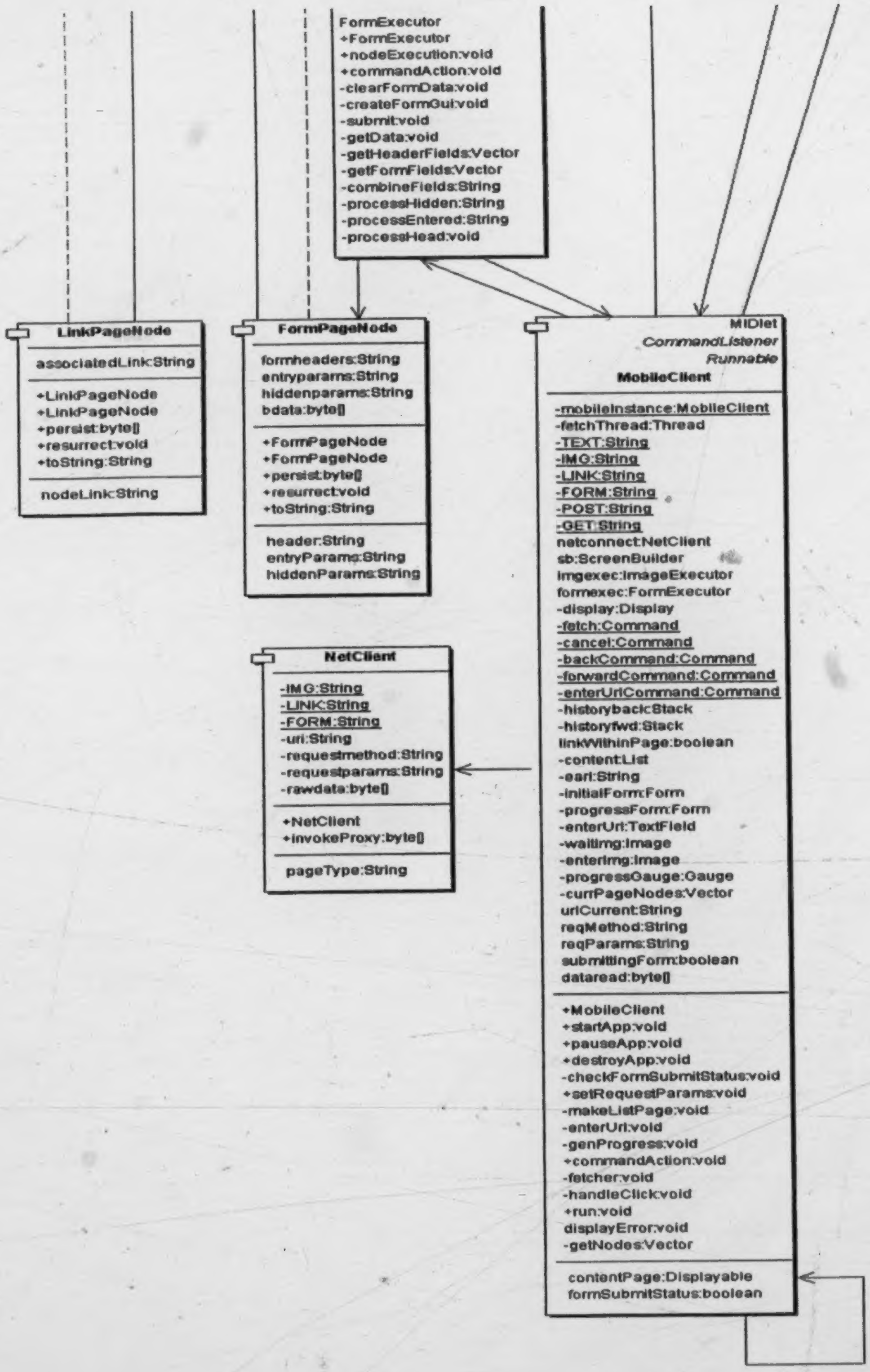
Note that the class diagrams shown in this section contain only the major classes used in the functionality for each component.

(A) The Mobile Client component

- Browser MIDlet: *MobileClient*
- Request Dispatcher: *NetClient*
- GUI Builder: *PageNode classes, ScreenBuilder, Executor classes*



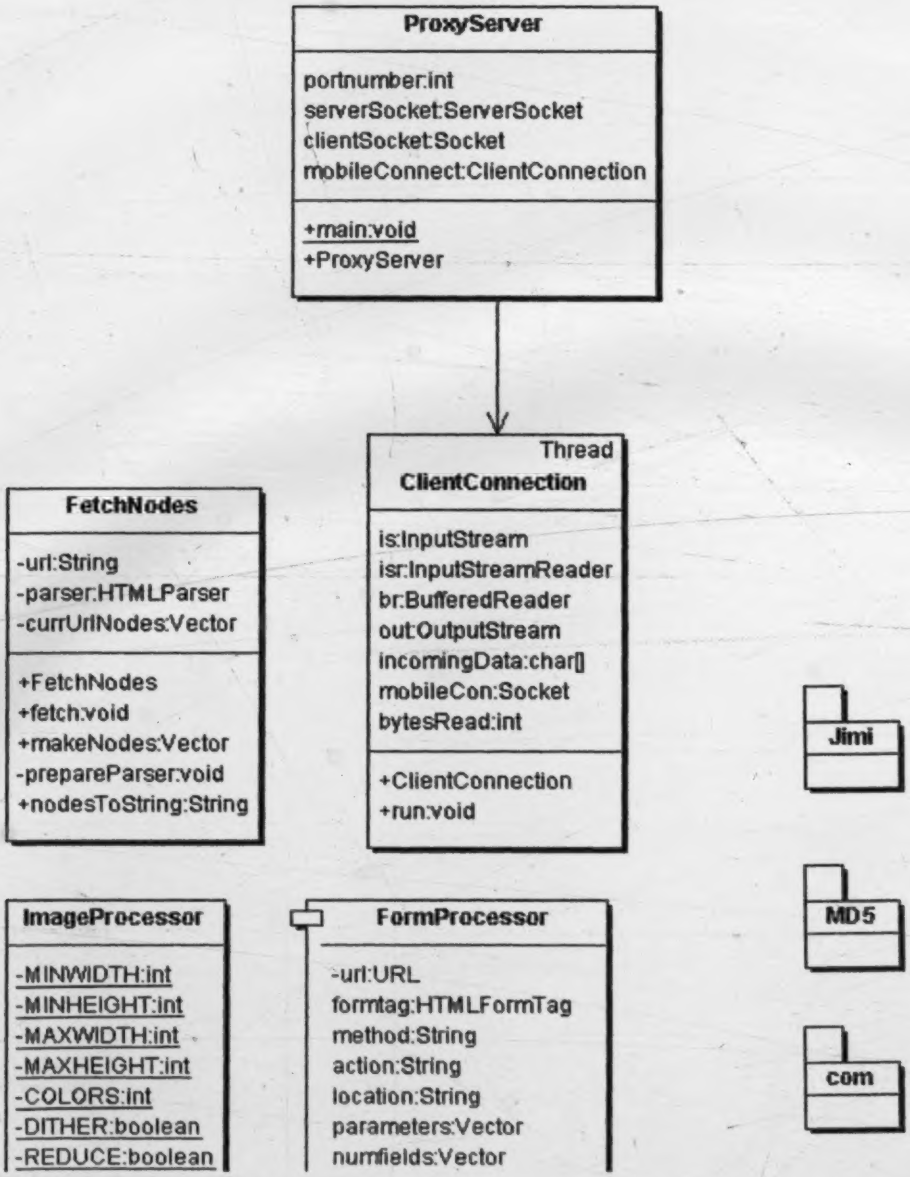
interface
Persistent
+persist:byte[]
+resurrect:void
VectorHelper
+persist:byte[]
+resurrect:Vector
Executor
+nodeExecution:void
PageNode
content:String
type:String
+PageNode
+PageNode
+persist:byte[]
+resurrect:void
+toString:String
nodeType:String
nodeContent:String
CommandListener
FormExecutor
-formDisplay:Form
-formSubmit:Command
-formClear:Command
-backFromForm:Command
-TEXT:String
-PWD:String
-STRING:String
-HIDDEN:String
-formAction:String
-formMethod:String
-page:List
-display:Display
-mobile:MobileClient
-formImage:Image
formnode:FormPageNode
head:String
fields:String
hiddenfields:String
allfields:String
method:String
action:String
location:String
formfields:Vector
ScreenBuilder
-holdnodes:Vector
screen:List
formtag:Image
linktag:Image
imgtag:Image
+ScreenBuilder
+createScreen:List
-createImages:void
CommandListener
ImageExecutor
-backFromImage:Command
-tempImage:Image
-page:List
-display:Display
-mobile:MobileClient
+ImageExecutor
+nodeExecution:void
+commandAction:void
ShowImage

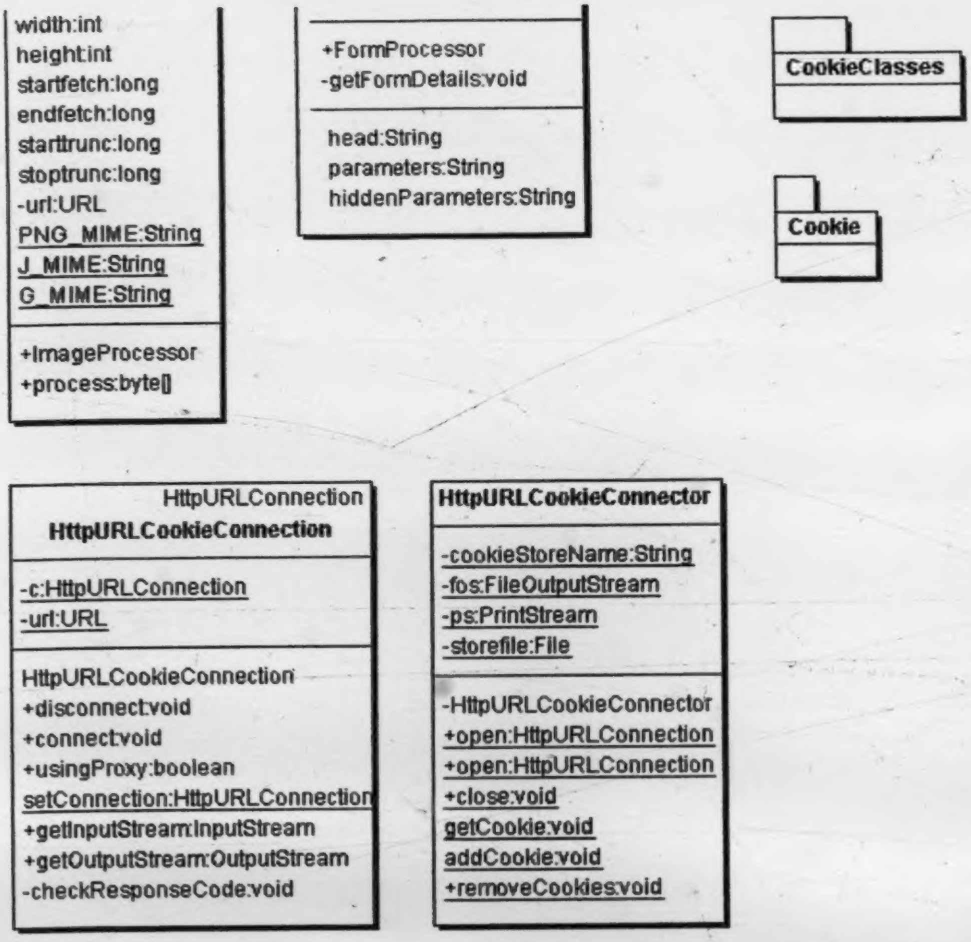

FormExecutor
+FormExecutor
+nodeExecution:void
+commandAction:void
-clearFormData:void
-createFormGui:void
-submit:void
-getData:void
-getHeaderFields:Vector
-getFormFields:Vector
-combineFields:String
-processHidden:String
-processEntered:String
-processHead:void
LinkPageNode
associatedLink:String
+LinkPageNode
+LinkPageNode
+persist:byte[]
+resurrect:void
+toString:String
nodeLink:String
FormPageNode
formheaders:String
entryparams:String
hiddenparams:String
bdata:byte[]
+FormPageNode
+FormPageNode
+persist:byte[]
+resurrect:void
+toString:String
header:String
entryParams:String
hiddenParams:String
MIDlet
CommandListener
Runnable
MobileClient
-mobileInstance:MobileClient
-fetchThread:Thread
-TEXT:String
-IMG:String
-LINK:String
-FORM:String
-POST:String
-GET:String
netconnect:NetClient
sb:ScreenBuilder
imgexec:ImageExecutor
formexec:FormExecutor
-display:Display
-fetch:Command
-cancel:Command
-backCommand:Command
-forwardCommand:Command
-enterUrlCommand:Command
-historyback:Stack
-historyfwd:Stack
linkWithinPage:boolean
-content:List
-earl:String
-initialForm:Form
-progressForm:Form
-enterUrl:TextField
-waitImg:Image
-enterImg:Image
-progressGauge:Gauge
-currPageNodes:Vector
urlCurrent:String
reqMethod:String
reqParams:String
submittingForm:boolean
dataread:byte[]
+MobileClient
+startApp:void
+pauseApp:void
+destroyApp:void
-checkFormSubmitStatus:void
+setRequestParams:void
-makeListPage:void
-enterUrl:void
-genProgress:void
+commandAction:void
-fetcher:void
-handleClick:void
+run:void
displayError:void
-getNodes:Vector
contentPage:Displayable
formSubmitStatus:boolean
NetClient
-IMG:String
-LINK:String
-FORM:String
-url:String
-requestmethod:String
-requestparams:String
-rawdata:byte[]
+NetClient
+invokeProxy:byte[]
pageType:String

(B) The Proxy Server component

(1) Processing a request at the proxy

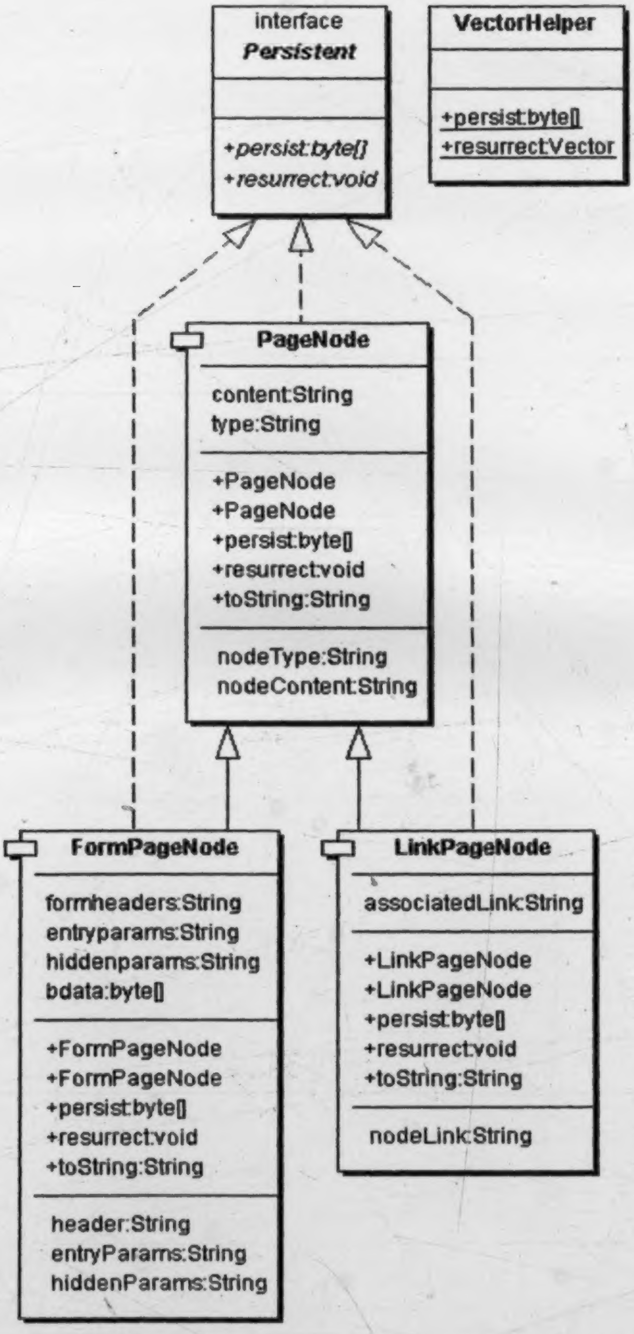
- Proxy Controller: P*roxyserver*
- Request Handling: *ClientConnection, FetchNodes, Processpr classes*
- Utility packages: *com (parsing), MD5 (hashing), CookieClasses (session handling), Jimi (Image processing).*


ProxyServer
portnumber:int
serverSocket:ServerSocket
clientSocket:Socket
mobileConnect:ClientConnection
+main:void
+ProxyServer
Thread
ClientConnection
is:InputStream
isr:InputStreamReader
br:BufferedReader
out:OutputStream
incomingData:char[]
mobileCon:Socket
bytesRead:int
+ClientConnection
+run:void
FetchNodes
-url:String
-parser:HTMLParser
-currUrlNodes:Vector
+FetchNodes
+fetch:void
+makeNodes:Vector
-prepareParser:void
+nodesToString:String
Jimi
MD5
com
ImageProcessor
-MINWIDTH:int
-MINHEIGHT:int
-MAXWIDTH:int
-MAXHEIGHT:int
-COLORS:int
-DITHER:boolean
-REDUCE:boolean
FormProcessor
-url:URL
formtag:HTMLFormTag
method:String
action:String
location:String
parameters:Vector
numfields:Vector


width:int
height:int
startfetch:long
endfetch:long
starttrunc:long
stoptrunc:long
-url:URL
PNG_MIME:String
J_MIME:String
G_MIME:String
+ImageProcessor
+process:byte[]
+FormProcessor
-getFormDetails:void
head:String
parameters:String
hiddenParameters:String
CookieClasses
Cookie
HttpURLConnection
HttpURLCookieConnection
-c:HttpURLConnection
-url:URL
HttpURLCookieConnection
+disconnect:void
+connect:void
+usingProxy:boolean
setConnection:HttpURLConnection
+getInputStream:InputStream
+getOutputStream:OutputStream
-checkResponseCode:void
HttpURLCookieConnector
-cookieStoreName:String
-fos:FileOutputStream
-ps:PrintStream
-storefile:File
-HttpURLCookieConnector
+open:HttpURLConnection
+open:HttpURLConnection
+close:void
getCookie:void
addCookie:void
+removeCookies:void

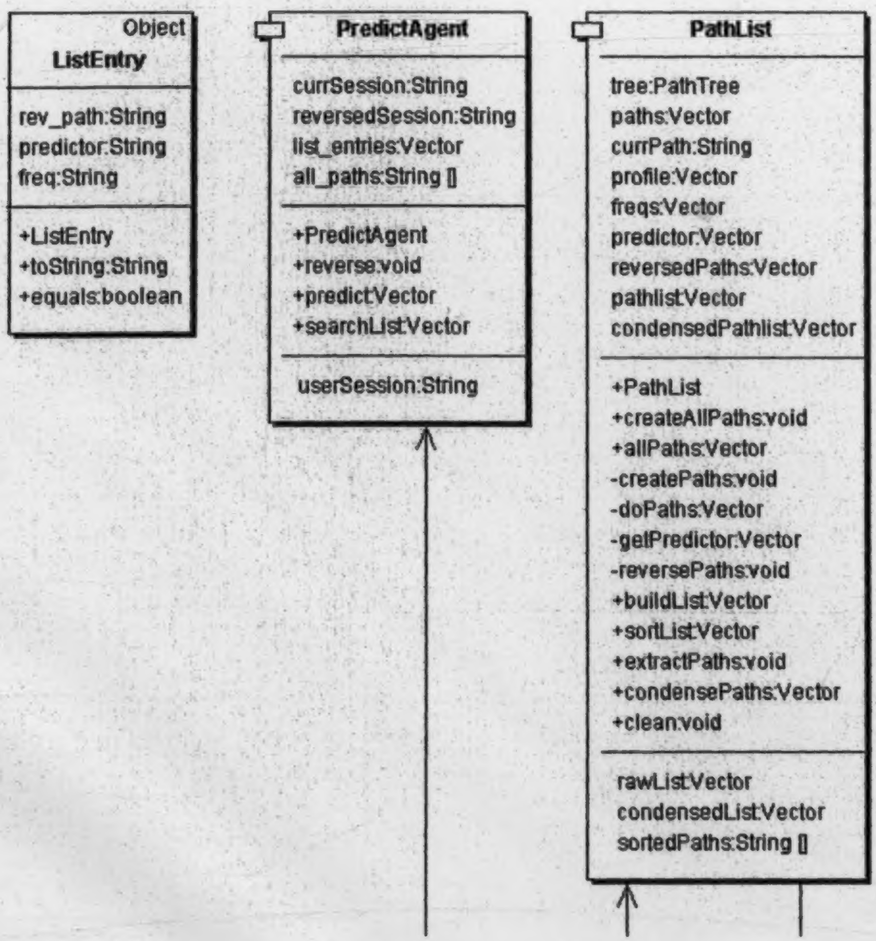
(2) Proxy Transcoder classes

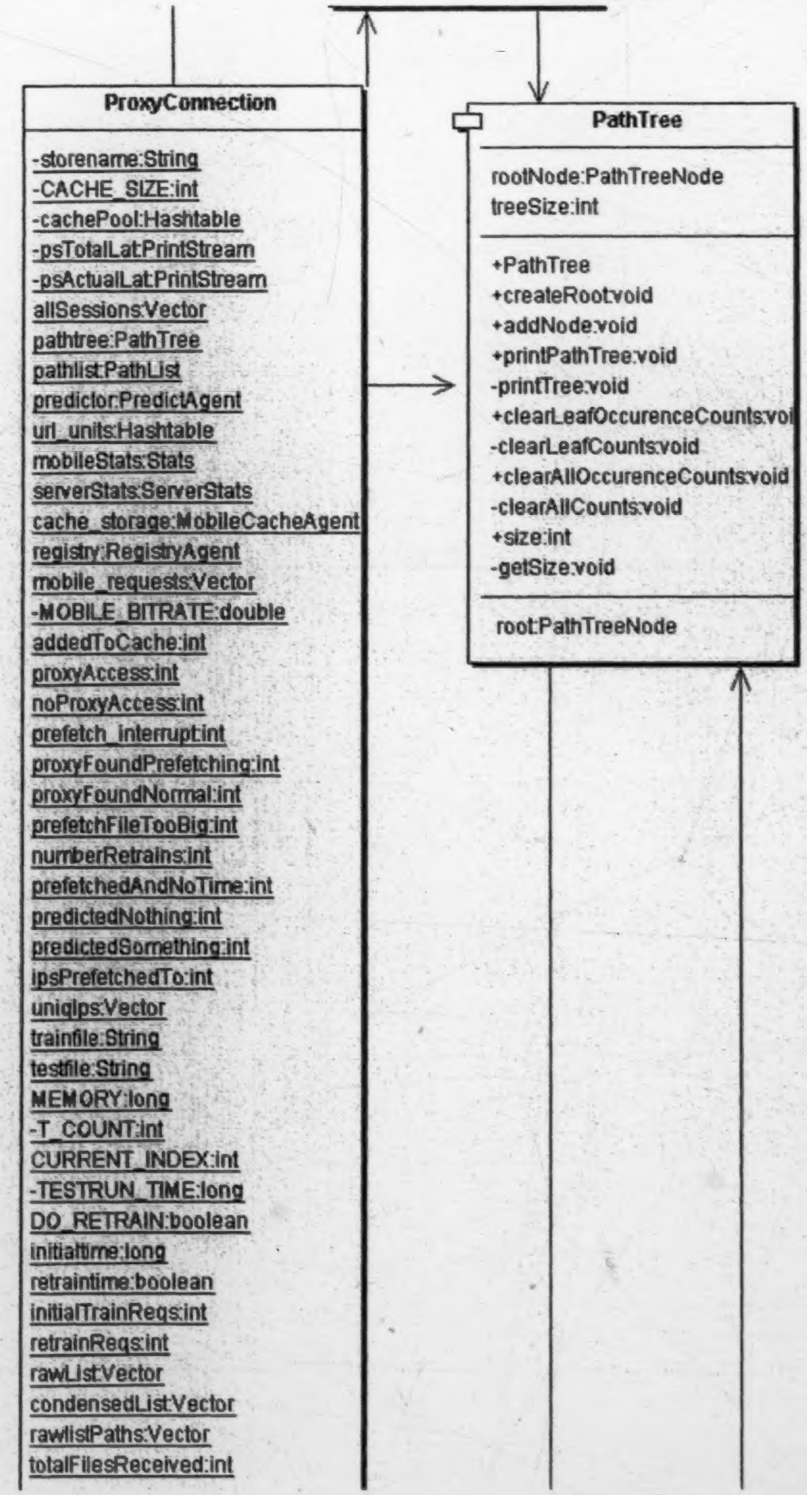
- PageNode package: *Small library of classes known to client and proxy.*
- Object Serialisation utility: *Persistent interface, VectorHelper class*

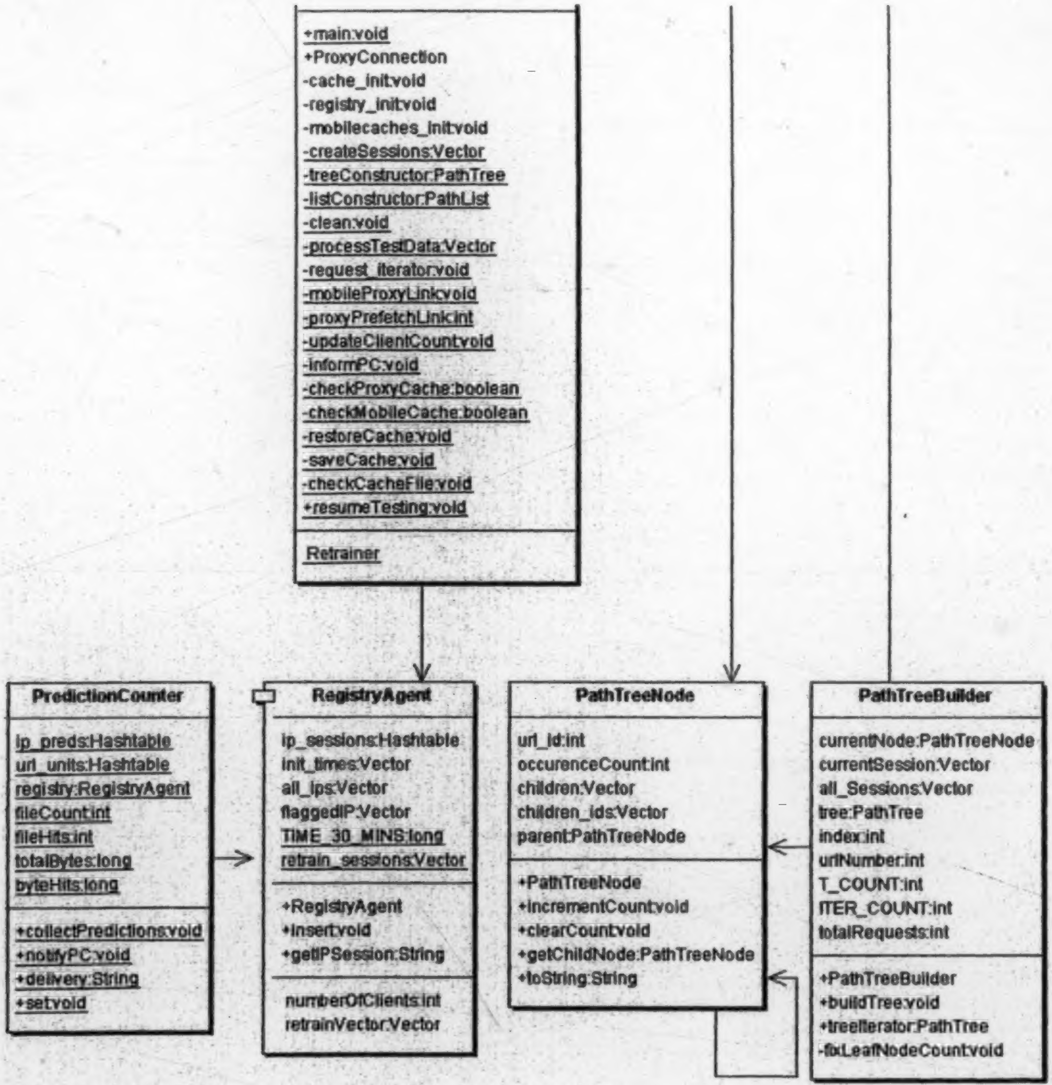

interface
Persistent
+persist:byte[]
+resurrect:void
VectorHelper
+persist:byte[]
+resurrect:Vector
PageNode
content:String
type:String
+PageNode
+PageNode
+persist:byte[]
+resurrect:void
+toString:String
nodeType:String
nodeContent:String
FormPageNode
formheaders:String
entryparams:String
hiddenparams:String
bdata:byte[]
+FormPageNode
+FormPageNode
+persist:byte[]
+resurrect:void
+toString:String
header:String
entryParams:String
hiddenParams:String
LinkPageNode
associatedLink:String
+LinkPageNode
+LinkPageNode
+persist:byte[]
+resurrect:void
+toString:String
nodeLink:String

(3) Prediction at the proxy

- Proxy Controller: *ProxyConnection class*
- Prediction Algorithm: *PathTree, PathTreeNode,PathTreeBuilder, PathList, ListEntry, PredictionCounter.*
- Session Tracker: *RegistryAgent.*

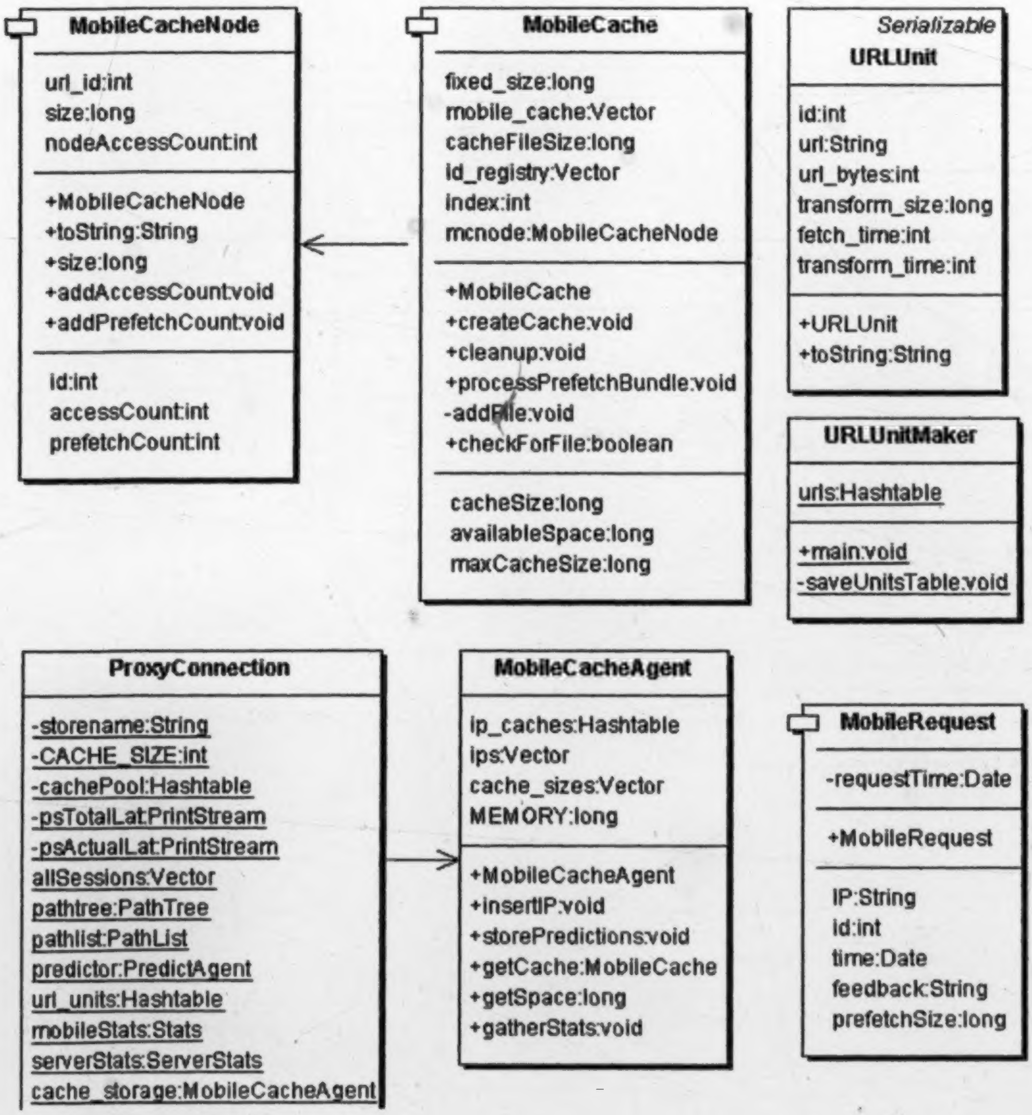

Object
ListEntry
rev_path:String
predictor:String
freq:String
+ListEntry
+toString:String
+equals:boolean
PredictAgent
currSession:String
reversedSession:String
list_entries:Vector
all_paths:String []
+PredictAgent
+reverse:void
+predict:Vector
+searchList:Vector
userSession:String
PathList
tree:PathTree
paths:Vector
currPath:String
profile:Vector
freqs:Vector
predictor:Vector
reversedPaths:Vector
pathlist:Vector
condensedPathlist:Vector
+PathList
+createAllPaths:void
+allPaths:Vector
-createPaths:void
-doPaths:Vector
-getPredictor:Vector
-reversePaths:void
+buildList:Vector
+sortList:Vector
+extractPaths:void
+condensePaths:Vector
+clean:void
rawList:Vector
condensedList:Vector
sortedPaths:String []

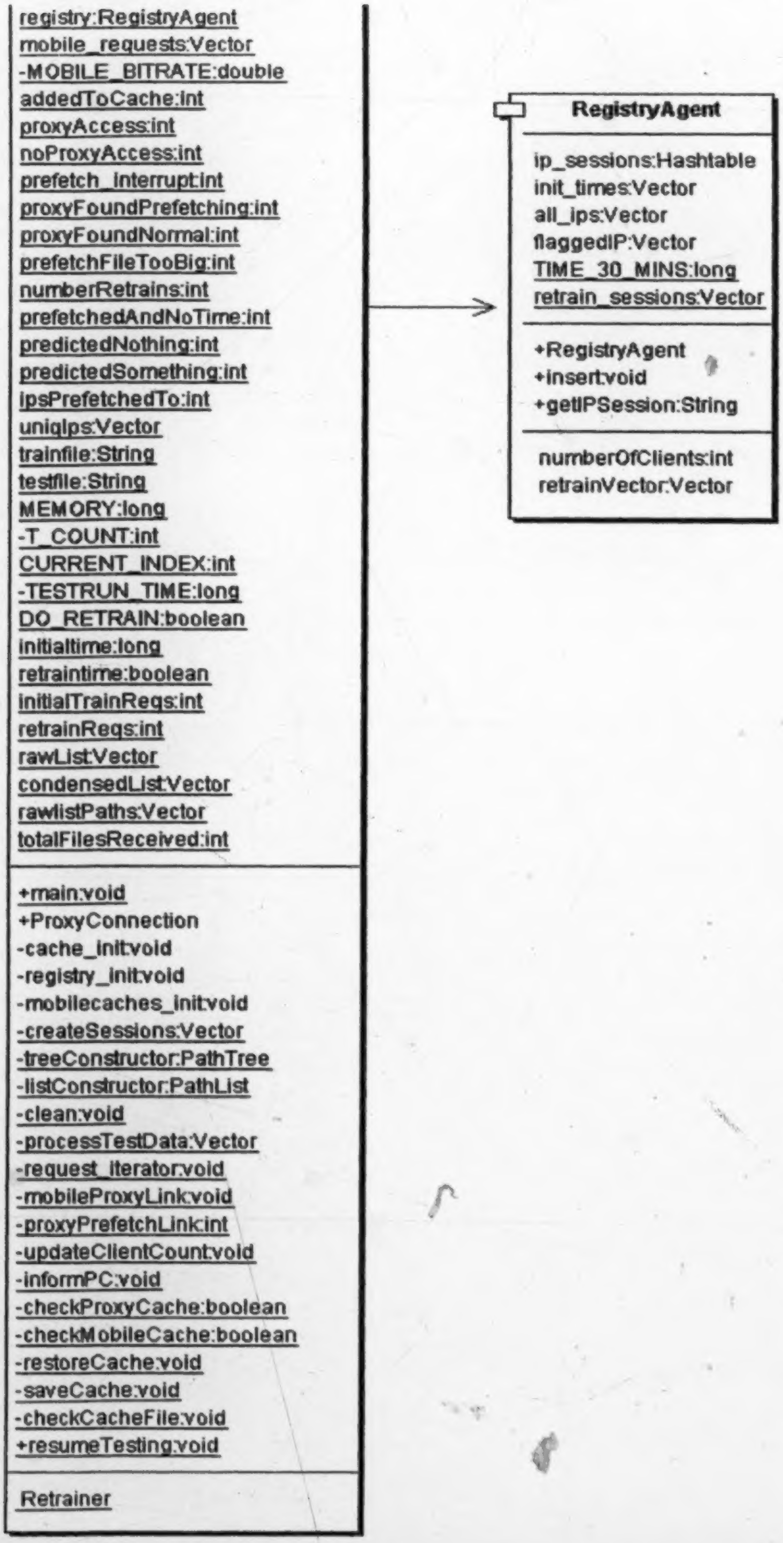

ProxyConnection
-storename:String
-CACHE_SIZE:int
-cachePool:Hashtable
-psTotalLat:PrintStream
-psActualLat:PrintStream
allSessions:Vector
pathtree:PathTree
pathlist:PathList
predictor:PredictAgent
url_units:Hashtable
mobileStats:Stats
serverStats:ServerStats
cache_storage:MobileCacheAgent
registry:RegistryAgent
mobile_requests:Vector
-MOBILE_BITRATE:double
addedToCache:int
proxyAccess:int
noProxyAccess:int
prefetch_interrupt:int
proxyFoundPrefetching:int
proxyFoundNormal:int
prefetchFileTooBig:int
numberRetrains:int
prefetchedAndNoTime:int
predictedNothing:int
predictedSomething:int
ipsPrefetchedTo:int
uniqIps:Vector
trainfile:String
testfile:String
MEMORY:long
-T_COUNT:int
CURRENT_INDEX:int
-TESTRUN_TIME:long
DO_RETRAIN:boolean
initialtime:long
retraintime:boolean
initialTrainReqs:int
retrainReqs:int
rawList:Vector
condensedList:Vector
rawlistPaths:Vector
totalFilesReceived:int
PathTree
rootNode:PathTreeNode
treeSize:int
+PathTree
+createRoot:void
+addNode:void
+printPathTree:void
-printTree:void
+clearLeafOccurenceCounts:voi
-clearLeafCounts:void
+clearAllOccurenceCounts:void
-clearAllCounts:void
+size:int
-getSize:void
root:PathTreeNode


+main:void
+ProxyConnection
-cache_init:void
-registry_init:void
-mobilecaches_init:void
-createSessions:Vector
-treeConstructor:PathTree
-listConstructor:PathList
-clean:void
-processTestData:Vector
-request_iterator:void
-mobileProxyLink:void
-proxyPrefetchLink:int
-updateClientCount:void
-informPC:void
-checkProxyCache:boolean
-checkMobileCache:boolean
-restoreCache:void
-saveCache:void
-checkCacheFile:void
+resumeTesting:void
Retrainer
PredictionCounter
ip_preds:Hashtable
url_units:Hashtable
registry:RegistryAgent
fileCount:int
fileHits:int
totalBytes:long
byteHits:long
+collectPredictions:void
+notifyPC:void
+delivery:String
+set:void
RegistryAgent
ip_sessions:Hashtable
init_times:Vector
all_ips:Vector
flaggedIP:Vector
TIME_30_MINS:long
retrain_sessions:Vector
+RegistryAgent
+insert:void
+getIPSession:String
numberOfClients:int
retrainVector:Vector
PathTreeNode
url_id:int
occurenceCount:int
children:Vector
children_ids:Vector
parent:PathTreeNode
+PathTreeNode
+incrementCount:void
+clearCount:void
+getChildNode:PathTreeNode
+toString:String
PathTreeBuilder
currentNode:PathTreeNode
currentSession:Vector
all_Sessions:Vector
tree:PathTree
index:int
urlNumber:int
T_COUNT:int
ITER_COUNT:int
totalRequests:int
+PathTreeBuilder
+buildTree:void
+treeIterator:PathTree
-fixLeafNodeCount:void

(4) Client Simulation at the Proxy server (Experiment 3)

- Client Cache Manager: *MobileCacheAgent*
- Client Cache handling: *MobileCacheNode, MobileCache.*
- Client request handling: *URLUnit classes, MobileRequest*
- Proxy Controller: *ProxyConnection*
- Session Tracker: *RegistryAgent*


MobileCacheNode
url_id:int
size:long
nodeAccessCount:int
+MobileCacheNode
+toString:String
+size:long
+addAccessCount:void
+addPrefetchCount:void
id:int
accessCount:int
prefetchCount:int
MobileCache
fixed_size:long
mobile_cache:Vector
cacheFileSize:long
id_registry:Vector
index:int
mcnode:MobileCacheNode
+MobileCache
+createCache:void
+cleanup:void
+processPrefetchBundle:void
-addFile:void
+checkForFile:boolean
cacheSize:long
availableSpace:long
maxCacheSize:long
Serializable
URLUnit
id:int
url:String
url_bytes:int
transform_size:long
fetch_time:int
transform_time:int
+URLUnit
+toString:String
URLUnitMaker
urls:Hashtable
+main:void
-saveUnitsTable:void
ProxyConnection
-storename:String
-CACHE_SIZE:int
-cachePool:Hashtable
-psTotalLat:PrintStream
-psActualLat:PrintStream
allSessions:Vector
pathtree:PathTree
pathlist:PathList
predictor:PredictAgent
url_units:Hashtable
mobileStats:Stats
serverStats:ServerStats
cache_storage:MobileCacheAgent
MobileCacheAgent
ip_caches:Hashtable
ips:Vector
cache_sizes:Vector
MEMORY:long
+MobileCacheAgent
+insertIP:void
+storePredictions:void
+getCache:MobileCache
+getSpace:long
+gatherStats:void
MobileRequest
-requestTime:Date
+MobileRequest
IP:String
id:int
time:Date
feedback:String
prefetchSize:long


registry:RegistryAgent
mobile_requests:Vector
-MOBILE_BITRATE:double
addedToCache:int
proxyAccess:int
noProxyAccess:int
prefetch_interrupt:int
proxyFoundPrefetching:int
proxyFoundNormal:int
prefetchFileTooBig:int
numberRetrains:int
prefetchedAndNoTime:int
predictedNothing:int
predictedSomething:int
ipsPrefetchedTo:int
uniqIps:Vector
trainfile:String
testfile:String
MEMORY:long
-T_COUNT:int
CURRENT_INDEX:int
-TESTRUN_TIME:long
DO_RETRAIN:boolean
initialtime:long
retraintime:boolean
initialTrainReqs:int
retrainReqs:int
rawList:Vector
condensedList:Vector
rawlistPaths:Vector
totalFilesReceived:int
+main:void
+ProxyConnection
-cache_init:void
-registry_init:void
-mobilecaches_init:void
-createSessions:Vector
-treeConstructor:PathTree
-listConstructor:PathList
-clean:void
-processTestData:Vector
-request_iterator:void
-mobileProxyLink:void
-proxyPrefetchLink:int
-updateClientCount:void
-informPC:void
-checkProxyCache:boolean
-checkMobileCache:boolean
-restoreCache:void
-saveCache:void
-checkCacheFile:void
+resumeTesting:void
Retrainer
RegistryAgent
ip_sessions:Hashtable
init_times:Vector
all_ips:Vector
flaggedIP:Vector
TIME_30_MINS:long
retrain_sessions:Vector
+RegistryAgent
+insert:void
+getIPSession:String
numberOfClients:int
retrainVector:Vector